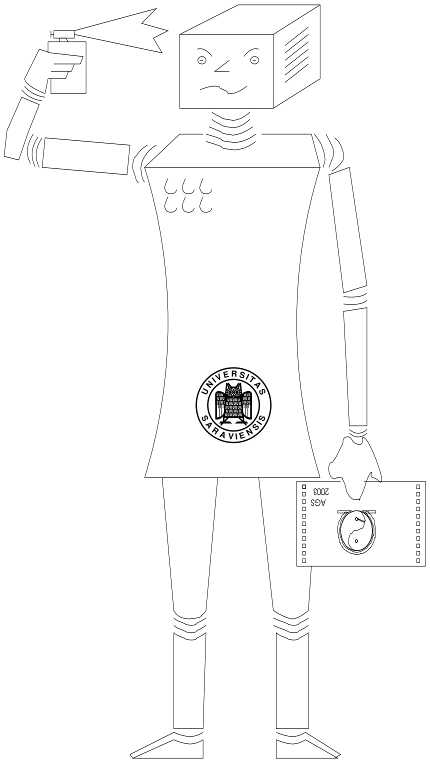
SEKI-REPORT ISSN 1437-4447

UNIVERSITÄT DES SAARLANDES
FACHRICHTUNG INFORMATIK
D–66123 SAARBRÜCKEN
GERMANY
WWW: http://www.ags.uni-sb.de/



The Proof Planners of ΩMEGA: A Technical Description

Andreas Meier
DFKI Gmbh, 66041 Saarbrücken, Germany
E-mail: ameier@dfki.de
WWW: http://www.ags.uni-sb.de/~ameier/

SEKI Report SR–2004–03

This SEKI Report was internally reviewed by:
Erica Melis
DFKI GmbH, 66041 Saarbrücken, Germany
E-mail: melis@dfki.de
WWW: http://www.ags.uni-sb.de/~melis/

Editor of SEKI series:
Claus-Peter Wirth
FR Informatik, Universität des Saarlandes, D–66123 Saarbrücken, Germany
E-mail: cp@ags.uni-sb.de
WWW: http://www.ags.uni-sb.de/~cp/welcome.html

The Proof Planners of ΩMEGA: A Technical Description

Andreas Meier
DFKI Gmbh, 66041 Saarbrücken, Germany
E-mail: ameier@dfki.de
WWW: http://www.ags.uni-sb.de/~ameier/

August 25, 2004

Abstract

The ΩMEGA proof development system employs proof planning for automated proof construction at the abstract level of methods. In this report we discuss the technical concepts underlying proof planning in ΩMEGA and give detailed descriptions of the algorithms of the two proof planners of the ΩMEGA system: PLAN which performs simple proof planning with methods and MULTI which performs multiple-strategy proof planning.

1 Introduction

Proof planning was originally conceived as an extension of tactical theorem proving to enable automated theorem proving at the abstract level of tactics. BUNDY's key idea in [9] is to augment individual tactics with pre- and postconditions. This results in planning operators, so-called *methods*. Thus, proof planning integrates both, elements from tactical theorem proving and elements from Artificial Intelligence (AI) planning. In the ΩMEGA [56] system the traditional proof planning approach is enriched by incorporating mathematical knowledge into the planning process (see [45]) and the introduction of the additional hierarchical level of strategies (see [44]).

Domain-specific knowledge can be encoded in *methods*, in *control rules*, and in *external systems* such as computer algebra systems or constraint solvers. Methods can encode not only general proving steps but also steps particular to a mathematical domain. Control rules enable meta-level reasoning about the current proof planning state as well as about the entire history of the proof planning process in order to guide the search.

The simple proof planner of the ΩMEGA system, which is called PLAN, searches at the level of methods, i.e., it searches for applicable methods and applies the instantiated methods. Case-studies revealed the drawbacks of the simple planner: It combines the application of methods, the instantiation of variables, and backtracking in a pre-defined way. Moreover, the functionalities of these subcomponents are very restricted. The hard-coded combination of operations with restricted functionalities prohibits the use of mathematical knowledge of certain proof constructions and their combination. As a result, the planner fails on problems for which more flexibility and knowledge is needed in the proof planning process (see [44, 34]).

These observations motivated the development of proof planning with multiple strategies. Proof planning with multiple strategies decomposes the previous monolithic proof planning process and replaces it by separate but collaborating operations, so-called strategies, which can realize different plan refinements and modifications, e.g., different kinds of step (i.e., instantiated method) computation and selection, different kinds of backtracking, different kinds of variable instantiation etc. Moreover, the decision on when to apply a strategy is not encoded once and forever into a fixed control procedure but rather is determined by meta-level reasoning using heuristic control knowledge of strategies and their combination. As compared with the previous proof planning, strategies and their heuristic control introduce another hierarchical level and can encode further (mathematical) domain knowledge. We realized proof planning with multiple strategies in the MULTI planner.

In this technical report, we describe the technical concepts underlying proof planning in ΩMEGA and give detailed descriptions of the algorithms of both planners of ΩMEGA. The structure of the report is as follows: We first describe the basics of knowledge-based proof planning in section 2, in particular, methods and control rules and the incorporation of external systems. In the subsequent section, we give a detailed description of PLAN including the discussion of a sample application. Section 4 introduces proof planning with multiple-strategies and gives a conceptual description of MULTI, which is complemented by a detailed technical description in section 5.

2 Basics of Proof Planning in ΩMEGA

Proof planning in ΩMEGA considers mathematical theorems as planning problems and combines tactical theorem proving and AI planning techniques. Hence, we start with a brief account of how theorem proving can be seen as an AI-planning problem comprising brief discussions of the background proof development in ΩMEGA and AI-planning. Next, we discuss tasks and the $\mathcal{PDS}$, which represent the current status during a proof planning process. Afterwards, we introduce ΩMEGA's method and control rule languages, describe the instantiations of methods, which are called actions, and briefly discuss the incorporation of external systems into proof planning.

Notation Functions that are part of the descriptions of methods, control rules, and algorithms are denoted with a *special font* (e.g., *term-at-position*). Since the core of ΩMEGA is implemented in LISP these functions are LISP functions in the implementation. For clarity, we write the application of the function *func* to the arguments $arg_1, \ldots, arg_2$ not in LISP syntax, i.e., $($*func* $arg_1 \ \ldots \ arg_n)$, but in prefix notation, i.e., *func*$(arg_1, \ldots, arg_n)$.

Notation We denote a *set of items* $it_1, \ldots, it_n$ with $\{it_1, \ldots, it_n\}$. A *list or sequence of items* (i.e., ordered set of items) $it_1, \ldots, it_n$ we write as $[it_1, \ldots, it_n]$. [] denotes the empty list. On sets the operations $\cup, \cap, -$ are defined as usual. On lists $\cup$ denotes the concatenation of lists. The result of $list_1 - list_2$ is $list_1$ without all elements that are in $list_2$. The operations *first*, *last*, *rest*, and *reverse* are defined on lists. The function *first* returns the first element of a list whereas the function *last* returns the last element of a list. The function *rest* returns the list that results from the deletion of the first element from the initial list. The function *reverse* returns a list whose elements are in the reverse order of the elements of the input list.

The set of all items it that satisfy a certain property $P(it)$ is written as $\{it|P(it)\}$. The analogous list is written as $[it|P(it)]$. The elements of such a list are ordered arbitrarily, if no order is explicitly specified.

Sets are denoted with symbols in calligraphic style (e.g., $\mathcal{M}$ for a set of methods and $\mathcal{C}$ for a set of control rules). Lists are denoted with symbols that are marked with an arrow as superscript (e.g., $\vec{A}$ for a sequence of actions).

2.1 From Theorem Proving to AI-Planning

Proof Development in ΩMEGA

The basic logic of the proof development system ΩMEGA is a higher-order variant of Gentzen's natural deduction (ND) calculus [23]. Similar to many interactive systems (c.f., NUPRL [14], ISABELLE [51], HOL [25], COQ [15], QUODLIBET [30]) ΩMEGA employs tactics for the construction of complex and more abstract proofs. The idea in tactical theorem proving [48] is that repeatedly occurring sequences of inference steps are encapsulated into macro steps, so-called tactics, which enable interactive proof construction at a higher level of abstraction.

Since tactic applications can be expanded to ΩMEGA's basic ND-calculus and can be combined with the application of ND-rules ΩMEGA needs a hierarchical proof data structure that represents a (partial) proof attempt at different levels of abstraction. This data structure is called the *proof plan data structure* $\mathcal{PDS}$ [12]. We will present the proof objects stored in a $\mathcal{PDS}$ in a linearized style with proof lines as introduced in [2]. A proof line is of the form $L.\ \Delta \vdash F\ (\mathcal{R})$, where L is a unique label, $\Delta \vdash F$ a sequent denoting that the formula F can be derived from the set of hypotheses Δ, and $(\mathcal{R})$ is a justification expressing how the line was derived. Lines that are not yet derived from other lines are called open lines and have an open justification. A line that is not open is called a *closed line*.

For instance, the initial $\mathcal{PDS}$ for the proof problem with theorem Thm and assumptions $Ass_1, \ldots, Ass_n$ is:

$L_{Ass_1}.$	L_{Ass_1}	$\vdash Ass_1$	(Hyp)
		$\vdots$	
$L_{Ass_n}.$	L_{Ass_n}	$\vdash Ass_n$	(Hyp)
$L_{Thm}.$	$L_{Ass_1}, \ldots, L_{Ass_n}$	$\vdash Thm$	$(Open)$

AI-Planning

An *AI-planning problem*[1] consists of

1. a description of the initial state of the world in some formal language,
2. a description of the agent's goals in some formal language, and
3. a description of the possible operations that the agent can performs in some formal language.

An *AI-planner* is an algorithm that is applied to a planning problem and returns a sequence of *actions*, i.e., instantiated operations, which will achieve the goal, when executed in any world satisfying the initial state description. Such a sequence of actions is also called a *solution plan*.

A very simple, yet very influential language is the STRIPS representation.[2] STRIPS describes the initial state of the world with a complete set of ground literals. It restricts the type of goals that may be specified to conjunctions of positive literals. Operations are represented in the STRIPS language as *operators* (also called *operator schemata*) with *preconditions* and *effects*. The preconditions of each operator have the same restriction as the problem's goals: they are a conjunction of positive literals. An operator's effects are a conjunction that may includes both, positive and negative literals. All the positive literals in the operator's effects are called the *add-list* of the operator, while all the negative literals are called the *delete-list* of the operator.

The classical approach to solve planning problems is *precondition achievement planning* [17]. Precondition achievement planning goes back to the General Problem Solver, GPS [50]. STRIPS focused and distilled the technique to the form used in planning: During the planning process, first an *unsatisfied precondition* is chosen (this condition is not true and but it should be). Then, the available operators are checked whether their add list contains an effect to achieve this precondition. One operator is chosen, appropriately instantiated (bind the variables of the operator to elements of the plan), and the resulting action is inserted into the plan under development. Then, the preconditions of the introduced action become new unsatisfied preconditions of the plan whereas the initially unsatisfied precondition is *satisfied* by an effect of the introduced action.

Theorem Proving as AI-Planning Problem

The *initial state* of a *proof planning problem* consists of the proof *assumptions* and the *goal description* consists of the *theorem*. Methods are the operators of proof planning, where methods are tactics known from tactical theorem proving augmented with pre- and postconditions in order to derive operators for AI-planning. Simple proof planning searches for a solution plan, i.e., a sequence of instantiated methods that transforms the initial state into a state in which the theorem holds. In order to find a solution plan, it searches for applicable methods and applies the instantiated methods. Similar to AI-planning we call the instantiation of a method (i.e., the instantiation of a proof planning operator) an *action*.

[1] See [64, 54] for introductions to AI-planning.

[2] The acronym "STRIPS" stands for "STanford Research Institute Problem Solver', a very famous and influential planner build in the 1970s to control an unstable mobile robot known as "Shakey" [21, 20].

2.2 Tasks and the $\mathcal{PDS}$

Central during the proof planning process in ΩMEGA are so-called tasks, which express the logical dependencies between goals and assumptions, and a $\mathcal{PDS}$, which represents the partial proof plan constructed so far. We shall now first explain the role of these two fundamental structures.

In AI-planning, an unsatisfied precondition in a plan under construction can be satisfied with a matching effect of any other action in the plan. In proof planning, however, this is not the case because of the logical context of open lines. Thus, ΩMEGA's proof planning uses so-called tasks to express which proof lines (closed and open) can be used to construct a subplan for an open line. A *task* is a pair $(L_{open}, \mathcal{SUPPS}_{L_{open}})$ where L_{open} is an open line and $\mathcal{SUPPS}_{L_{open}}$ is a set of lines. The first element of a task is called the *task line* or the *goal of the task* and the second element is called the *support lines* or *supports*. The formula of the goal is also called *task formula*. A task with goal L_{open} and supports $\mathcal{SUPPS}_{L_{open}}$ is written as $L_{open} \blacktriangleleft \mathcal{SUPPS}_{L_{open}}$. During the planning process a list of all current tasks is stored in a so-called *agenda*. For a problem with theorem Thm and assumptions $Ass_1, \ldots, Ass_n$ the *initial agenda* consists of the task $L_{Thm} \blacktriangleleft \{L_{Ass_1}, \ldots, L_{Ass_n}\}$ where L_{Ass_i} and L_{Thm} are the proof lines of the initial $\mathcal{PDS}$ of the problem.

As example for the necessity to maintain a separate set of supports for each goal consider the introduction of a case-split. Let a goal $F[x]$ have the support line $x > 0 \vee x \leq 0$.[3] The introduction of a case-split results in two branches with: subtask $F[x] \blacktriangleleft \{x > 0, \ldots\}$ and $F[x] \blacktriangleleft \{x \leq 0, \ldots\}$. It would be incorrect, if the second subtask used the first assumption or vice versa. Moreover, actions can remove support lines of a task such that afterwards the planner cannot use these lines anymore. This is sensible, for instance, when an action simplifies a given support line with formula $x + 0 > 0$ to the new support with formula $x > 0$. Likely, the old support will not be needed anymore.

The proof plan under construction is represented in a $\mathcal{PDS}$. The *initial* $\mathcal{PDS}$ consists of the lines L_{Thm} and $L_{Ass_1}, \ldots, L_{Ass_n}$. The effects and the preconditions of actions in ΩMEGA's proof planning are proof lines. When a new action is added, then the new lines derived by this action are added into the $\mathcal{PDS}$. Moreover, all effect lines of the action are justified by an application of the method of the action to the premises of the action. For instance, if an action of method M has the premise lines L_1 and L_2 and the effect line L_3, then L_3 becomes justified in the $\mathcal{PDS}$ by $(M\ L_1\ L_2)$.

The justifications of the proof lines in the constructed $\mathcal{PDS}$ comprise the same information as causal links known from partial-order planning (see [64, 54]): which preconditions of an action are satisfied by which effects of other actions and – vice versa – which effects of an action are used to satisfy which preconditions of other actions. Thus, the $\mathcal{PDS}$ stores information such as which lines are used by actions and which lines depend on which other lines. Moreover, it keeps track of all proof lines created so far. Thereby, open lines in the $\mathcal{PDS}$ represent unsatisfied preconditions of actions (initially, the theorem) whereas closed lines are effects of actions (initially, the proof assumptions).

During a proof planning process, tasks in the agenda do always correspond to open lines in the $\mathcal{PDS}$, that is, for an open line in the current $\mathcal{PDS}$ there exists a task in the current agenda with this line as goal and vice versa. Thus, with respect to the agenda and the constructed $\mathcal{PDS}$,

[3] To simplify this example, we just write the formulas of the goal and the support line instead of the whole proof lines.

we can state the aim of the proof planning process as follows: Compute a sequence of actions, which derives, starting from the initial agenda and the initial $\mathcal{PDS}$, an empty agenda and a *closed* $\mathcal{PDS}$, that is, a $\mathcal{PDS}$ without open lines. The *solution proof plan* is a record of this sequence of actions. The simultaneous achievement of an empty agenda and a closed $\mathcal{PDS}$ mirrors the two roots of proof planning: From the AI-planning point of view the aim is to compute a sequence of actions that satisfy all goals, that is, to reach an empty agenda. From the tactical theorem proving point of view the aim is to apply a sequence of tactics, which result in a closed $\mathcal{PDS}$.

The proof planners PLAN and MULTI essentially work on an agenda and its tasks. First, they compute applicable actions for the current tasks. Then, they select one action and apply it. This results in new tasks. Technically, the simultaneous maintenance of a $\mathcal{PDS}$ during the proof planning process is not necessary for the two planners. In particular, if needed, a closed $\mathcal{PDS}$ could be constructed from the computed set of actions later on. However, historically proof planning in ΩMEGA did construct a $\mathcal{PDS}$ and an agenda was only introduced as a bookkeeping mechanism for the open proof lines. Practically, the $\mathcal{PDS}$ is important because of two reasons: First, ΩMEGA's tools for user interaction (e.g., its graphical user interface) are based on the $\mathcal{PDS}$ as the central data structure. During the proof planning process the constructed $\mathcal{PDS}$ is presented to the user as the current state of progress. When describing sample applications of PLAN and MULTI in section 3.5 and 4.5 we shall also use $\mathcal{PDS}$s as a means to display and discuss the constructed proof plans. Second, the $\mathcal{PDS}$ is a representation of the current proof plan, i.e., the current sequence of actions, and explicitly stores information that is important for the control rules (e.g., which lines depend on which other lines etc.). Although this information could be computed from the current sequence of actions each time it is needed, it is more convenient to use the $\mathcal{PDS}$ as a bookkeeper.

2.3 Methods

Methods encode the knowledge of the relevant proof steps of mathematical domains. Technically, a method in ΩMEGA is a frame data structure with the slots **declarations**, **parameters**, **application conditions**, **premises**, **conclusions**, **outline computations**, **expansion computations**, and **proof schema**.

The *premises and conclusions of a method* specify the preconditions and the effects of the method.[4] The conclusions should be logically inferable from the premises. The union of conclusions and premises is called the *outline* of a method. Declarative descriptions of the formulas of the outline can be given in the proof schema, which also provides the schematic or procedural expansion information (see below).

Premises and conclusions may be annotated with $\oplus$ and $\ominus$. The annotations are needed to indicate whether a method is used for forward or backward search. As opposed to AI-planning, where operators typically can be applied for both forward search and backward search, a method in ΩMEGA is either used in forward search or in backward search. This is because methods typically comprise complex computations that are reasonable either in one direction or in the other direction.

As example, consider methods that employ a computer algebra system to simplify numerical expressions. A backward method can employ the computer algebra system in order to reduce

[4] That preconditions and effects of a method are called the premises and conclusions of the method, respectively, is an example for the combination of AI-planning and tactical theorem proving in proof planning. If we see the method as tactic, then the effects of a method are the conclusions of a tactic and the preconditions are the premises.

a goal to a simplified goal. A corresponding forward method can employ the computer algebra system in order to derive a simplified support line. But what should the backward method perform when applied forwards? Does it obtain a "simplified" support line and tries to "complicate" it in order to obtain a more "difficult" support? Vice versa, what should the forward method perform when applied backwards? Does it obtain a "simplified" goal, which it tries to "complicate"?

Backward and forward methods are specified as follows: A *backward method* has $\ominus$ *conclusions* and $\oplus$ *premises* as well as $\ominus$ *premises* and *blank premises*. To compute an action of the method, one of the $\ominus$ conclusions is matched with the goal of a given task and both, the $\ominus$ premises and the blank premises, are matched with supports of the task. When the resulting action is introduced into the proof plan, then the goal is closed in the $\mathcal{PDS}$ and the $\oplus$ premises are added to the $\mathcal{PDS}$ and become goals of new tasks. These new tasks inherit the supports of the initial task except that the $\ominus$ premises are removed. The blank premises are not affected. A *forward method* has $\oplus$ *conclusions* as well as $\ominus$ premises and blank premises. To compute an action of the method, the $\ominus$ premises and the blank premises are matched with the support lines of a given task. When the resulting action is introduced into the proof plan, then the $\oplus$ conclusions are added to the $\mathcal{PDS}$ and become new support lines of the task. Moreover, the $\ominus$ premises are removed from the supports of the task. Again, the blank premises are not affected.

Method: =Subst-B	
declarations	type-variables: α variables: f_o, f'_o, t_α, t'_α, $pos_{position}$ tf_α, tf'_α, $\lambda f_{\alpha o}$
parameters	*pos*
appl. conds.	(1) *valid-position-p*(f, pos) (2) [*term-at-position*$(f, pos) = t \vee$ *term-at-position*$(f, pos) = t'$]
premises	$\oplus L_2$, L_1
conclusions	$\ominus L_3$
outline computations	$f' \leftarrow$ *replace-at-position*(f, t, t', pos)
expansion computations	$tf \leftarrow$ *term-at-position*(f, pos) $tf' \leftarrow$ *term-at-position*(f', pos) $\lambda f \leftarrow$ *lambda-abstraction*(f, pos)
proof schema	$L_1.\quad \Delta \quad \vdash t \doteq t' \quad ()$ $L_2.\quad \Delta \quad \vdash f' \quad (Open)$ $L_4.\quad \Delta \quad \vdash \forall P_{\alpha o}\text{.}\, P(tf') \Rightarrow P(tf) \quad (\equiv_E \;\; \doteq)$ $L_5.\quad \Delta \quad \vdash (\lambda f)(tf') \Rightarrow (\lambda f)(tf) \quad (\forall_E \;\; L_4 \;\; \lambda f)$ $L_6.\quad \Delta \quad \vdash f[tf'] \Rightarrow f[tf] \quad (\lambda\leftrightarrow \;\; L_5)$ $L_3.\quad \Delta \quad \vdash f \quad (\Rightarrow_E \;\; L_2 \;\; L_6)$

Figure 1: The =Subst-B method.

Consider the method =Subst-B, given in Figure 1, which can be used in all domains that employ the equality $\doteq$. Essentially, the method performs an equality substitution. It has two preconditions L_1 and L_2, where the proof schema determines L_1 to be an equation. The only conclusion is L_3. =Subst-B is a backward method. The introduction of an action of =Subst-B closes a task line whose formula matches with the formula of L_3 and introduces a new task whose goal is the instantiation of L_2. That is, the formula of the new goal results from the formula of the initial goal by substitution with the equation, which is the formula of a support of the initial task that matched with L_1. For instance, =Subst-B applied to the task *even*$(a+1) \blacktriangleleft \{a = 1, \ldots\}$[5] introduces the new goal *even*$(1+1)$.

In the *declarations of a method* the variables of the method and their types are introduced.

The *parameters of a method* are specific variables that influence the resulting action, when the method is instantiated. The =Subst-B method has the parameter *pos* which is of type *position*. The method can be applied to different positions, e.g., for the task $even(a+a) \blacktriangleleft \{a = 1, \ldots\}$ at the first or the second occurrence of a in the goal. The choice of *pos* determines which a should be replaced.

The *application conditions of a method* are meta-level descriptions that restrict the applicability of a method. The application conditions can consist of arbitrary LISP functions. The method =Subst-B has two application conditions: (1) the position *pos* has to be a valid position in the formula f and (2) the subterm in f at the position *pos* is t or t'. Note that application conditions reason only about whether the application of a method is valid in a certain situation; they do not reason about whether the application is useful.

The *outline computations of a method* allow to apply arbitrary LISP functions to compute the new terms and formulas of new outline lines generated by an application of the method. The outline computation of =Subst-B specifies that the new formula f' is computed from f by replacing t by t' or t' by t at the position *pos* depending on whether the subterm in f at position *pos* is t or t'.

Similarly, the *expansion computations of a method* allow to apply arbitrary LISP functions to compute the new terms and formulas generated during the expansion of an action of the method. The expansion computation of =Subst-B specifies that the terms tf and tf' are computed as the subterms of f and f' at position *pos*, respectively. Moreover, the term λf is computed as a λ-abstraction of f where the term at position *pos* is replaced by the λ -bound variable (that is, essentially λf has the form $\lambda x_\alpha.f[x]$, where $f[x]$ is the term that results from f by replacing the subterm at position *pos* by x).

The *proof schema of a method* is a declarative description of the outline of a method and of the expansion of actions of the method. Expansions of actions corresponds to both tactic expansions and expansions of HTN-planning [60]. When an action of the method is expanded, then for each conclusion a new subproof is introduced into the $\mathcal{PDS}$ resulting in new justifications of the conclusion at a lower level of abstraction. For instance, the proof schema of =Subst-B specifies that the defined concept $\doteq$ in the premise is replaced by its definition. Then, the calculus rules $\forall_E$, $\lambda\leftrightarrow$, and $\Rightarrow_E$ are applied to derive the conclusion of the method.

Generally, proof construction may require to construct mathematical objects, for instance, if a method has to instantiate existentially quantified variables by witness terms. A witness term has to be a concrete term. However, if the method is applied at an early stage of the proof, the planner generally has no knowledge of the true nature of the witness term. Therefore, the actual instantiation of witnesses is postponed; rather, methods introduce so-called *meta-variables* as temporary substitutes for the actual witness terms, which will be determined at a later point in the planning process and subsequently instantiated.

[5] To simplify this example, we just write the formulas of the goal and the support line instead of the whole proof lines.

Notation In this report, we write mv for meta-variables. If several meta-variables occur, we attach subscripts to mv in order to distinguish the meta-variables. We either use the variable for whose instantiation the meta-variable is a substitute as subscript (e.g., we write mv_x if mv is a substitute for the instantiation of the variable x) or we use numbers. If the decomposition of a quantified formula results in the introduction of a constant, then we write c for this constant. Similar to the notation for meta-variables, we use either the initial variable or numbers as subscripts to distinguish several occurring constants.

Notation Methods are written in SMALL CAPITAL FONT (e.g., =Subst-B). The name of backward methods ends with -B whereas the name of forward methods ends with -F.

2.4 Actions

An action is an instantiation of a method. Technically, an *action* in ΩMEGA is a frame data structure that has the slots **method**, **task**, **premises**, **conclusions**, **binding**, and **constraints**. The *method of an action* is a pointer to the method of which the action is an instantiation. The *task of an action* is a pointer to the task with respect to which the action was computed. The *conclusions and premises of an action* are sets of proof lines, respectively, which can be annotated with $\ominus$ and $\oplus$. The *binding of an action* is a substitution that (1) maps outline lines of the method to proof lines and (2) maps variables specified in the declarations of the method to terms, positions, etc. The *constraints of an action* are constraints that can be created by the evaluation of the application conditions of a method and that have to be passed to external constraint solvers (see section 2.6). Similar to methods, we call the union of the premises and conclusions of an action the *outline* of the action. The union of $\oplus$ premises and $\oplus$ conclusions of an action is also called the *new lines* of an action (i.e., the proof lines which are produced by an action), whereas the union of $\ominus$ premises, blank premises, and $\ominus$ conclusions is called the *given lines* of an action (i.e., the proof lines which have to be given in order to compute an action). Actions of forward methods are also called *forward actions* whereas actions of backward methods are also called *backward actions*.

Example 2.1

Action	
method	=Subst-B
task	$L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$
premises	$\oplus\ L_{Thm'}.\ L_{Ass_1}, L_{Ass_2}\ \vdash\ even(c+b)\ (Open)$ $L_{Ass_1}.\ L_{Ass_1}\ \vdash\ a \doteq c\ (Hyp)$
conclusions	$\ominus\ L_{Thm}.\ L_{Ass_1}, L_{Ass_2}\ \vdash\ even(a+b)\ (Open)$
binding	$\{L_3 \rightarrow L_{Thm}, L_1 \rightarrow L_{Ass_1}, L_2 \rightarrow L_{Thm'}, f \rightarrow even(a+b), \alpha \rightarrow \nu,$ $t \rightarrow a, t' \rightarrow c, pos \rightarrow < 1\ 1 >, f' \mapsto even(c+b)\}$
constraints	$\emptyset$

Figure 2: An action with the =Subst-B method.

Consider the action in Figure 2. It is an instantiation of the method =Subst-B computed with respect to the task $L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$. The proof line L_{Thm} is the only conclusion of the action (annotated with $\ominus$) whereas the proof lines L_{Ass_1} and $L_{Thm'}$ are the premises of the action ($L_{Thm'}$ annotated with $\oplus$). The binding maps all outline lines of the =Subst-B method (i.e., L_1, L_2, L_3) to the conclusions and the premises of the action and maps all variables declared in =Subst-B to terms and positions. The constraints of this action are empty.

The instantiation of a method in order to compute an admissible action comprises the following steps: First, the formulas of the conclusions and premises have to be matched with formulas of goals and their supports. If this succeeds, then the application conditions can be evaluated. If they evaluate to true, the method is applicable (wrt. to the computed matchings). Then, the outline computations have to be performed and the new lines of the outline have to be computed to complete the action. A detailed description on how actions are computed, selected, and introduced into a proof plan is given in the next section, when we describe PLAN. For the action in Figure 2 we give a summary of the computation and introduction into a proof plan here.

Suppose the current $\mathcal{PDS}$ corresponding to the task $L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$ is:

L_{Ass_1}.	L_{Ass_1}	$\vdash a_\nu \doteq c_\nu$	(Hyp)
L_{Ass_2}.	L_{Ass_2}	$\vdash b_\nu \doteq c$	(Hyp)
L_{Thm}.	L_{Ass_1}, L_{Ass_2}	$\vdash even_{\nu o}(a+b)$	$(Open)$

When the action in Figure 2 is computed, then first the lines L_1 and L_3 of the method =Subst-B are matched with the lines L_{Ass_1} and L_{Thm} of the $\mathcal{PDS}$, respectively. Afterwards, the application conditions are evaluated and the outline computations of the method are performed. Next, the missing outline is computed. In our example, the new $\oplus$ premise $L_{Thm'}$ is computed and is justified with $Open$. When the action is introduced, then its effect L_{Thm} is justified in the $\mathcal{PDS}$ by an application of the method =Subst-B to the premises $L_{Thm'}$ and L_{Ass_1} of the action. Moreover, the new proof line $L_{Thm'}$ is introduced into the $\mathcal{PDS}$. The resulting $\mathcal{PDS}$ is:

L_{Ass_1}.	L_{Ass_1}	$\vdash a_\nu \doteq c_\nu$	(Hyp)
L_{Ass_2}.	L_{Ass_2}	$\vdash b_\nu \doteq c$	(Hyp)
$L_{Thm'}$.	L_{Ass_1}, L_{Ass_2}	$\vdash even(c+b)$	$(Open)$
L_{Thm}.	L_{Ass_1}, L_{Ass_2}	$\vdash even_{\nu o}(a+b)$	(=Subst-B $L_{Thm'}$ L_{Ass_1})

The task $L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$ is then replaced by the task $L_{Thm'} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$ in the agenda.

Proof planning in ΩMEGA is a process that computes actions and introduces them into the proof plan under construction. However, since the introduced actions are represented in the $\mathcal{PDS}$ as applications of their methods we also use the phrase *action application* instead of action introduction, if we want to emphasize the changes in the $\mathcal{PDS}$. We also use the following vocabulary from tactical theorem proving. We say that the application of a backward action *closes an open line or a task*, if the open line or the goal of the task is an effect of the action and is closed by the introduction of the action into the proof plan under construction. We say that a *forward action is applied to some lines or to some supports*, if the lines or supports are the preconditions of the action. Moreover, we say that we *apply a method to a task or to some lines* as an abbreviation for the application of an action of the method to the task or to some lines.

2.5 Control Rules

Control rules provide guidance of the proof planning process by declaratively representing heuristical knowledge that corresponds to mathematical intuition about how to prove a goal in a certain situation. In particular, these rules provide the basis for meta-level reasoning and a global guidance since they can express conditions for a decision that depends on all available knowledge about the proof planning process so far. Several experiments indicate the superiority of a separate representation of control knowledge by control rules [49]. This representation is well-suited for modifications and for learning. The control rules used in Ωmega's proof planning were adopted from the control rule approach of the AI-planner Prodigy [62],

In the planning process control rules guide decisions at choice points, e.g., which task to tackle next or which method to apply next. They achieve this by reasoning about the heuristic utility of different alternatives[6] in order to promote the alternatives that seem to suit best in the current situation, where 'situation' comprises all available information on the current status such as the current tasks, their supports, the planning history, failed attempts etc. To manipulate an alternative list control rules can remove elements, prefer certain elements, or add new elements. This way, the ranking of alternatives is dynamically changed. This can help to prune the search space or to promote certain promising search paths.

Technically, control rules consist of an IF- and a THEN-part. The IF-part is a predicate on the current proof planning 'situation', whereas in the THEN-part modifications of alternative lists are stated. Moreover, each control rules specifies its kind, i.e., the choice point in the proof planning process it guides.

```
(control-rule prove-inequality
              (kind methods)
              (IF (and (goal-matches (REL A B))
                       (in REL {<,>,≤,≥})))
              (THEN (prefer (TellCS-B TellCS-F AskCS-B Simplify-B
                             Simplify-F Solve*-B ComplexEstimate-B
                             SetFocus-B))))
```

Figure 3: The control rule `prove-inequality`.

Figure 3 gives as example the control rule `prove-inequality`, which is evaluated during the selection of the next method to apply. In its IF-part `prove-inequality` checks whether the current goal is an inequality. If this is the case, it prefers the methods TellCS-B, TellCS-F, AskCS-B, Simplify-B, Simplify-F, Solve*-B, ComplexEstimate-B, and SetFocus-B in this order (these methods are explained in section 2.6 and section 3.5). The *prefer* states that the methods specified in the control rule are preferred before all other methods, i.e., the specified methods are ordered in front of the resulting alternative list. Other possible modifications of alternative lists are *select*, *reject*, *defer*, and *order-in-front*. *select* states that all other methods except those specified in the control rule are eliminated from the list of alternative methods. *reject* removes all alternatives specified in the control rule from a given alternative list, the latter

[6] As opposed to application conditions of methods, which reason about the legal feasibility of applications of methods (see last section).

two manipulations reorder the alternative list. *defer* orders all specified alternatives at the end of the alternative list, and *order-in-front* orders specified alternatives in front of other specified alternatives. Finally, there is the *insert* modification. It allows to introduce new elements in an alternative list. A typical situation for using an *insert* control rule is when a general control rule – which is applied first – removes some elements from the alternative list, which are needed in a particular situation. Then a more specific *insert* control rule, which is applied later on, can introduce the needed elements again.

Notation Control rules are denoted in the `typewriter font` (e.g., `prove-inequality`). Technically, control rules are frame data structures. Since they are considerably simpler as, for instance, methods, we do not present them in the data structure fashion (as we do with methods) rather we give their LISP encoding. That is, the content of Figure 3 is the specification of the control rule `prove-inequality` as it is in ΩMEGA's data base.

2.6 Incorporating External Systems into Proof Planning

We use a special kind of domain knowledge in ΩMEGA, namely the knowledge about and in external "expert" systems. Proof problems usually require many different capabilities for their solution, for instance, computation and object construction. In order to solve problems, it is often necessary to access several systems with complementary capabilities and to make use of their results. Various "expert" systems exist for mathematical problem solving, which have their specific data structures and very efficient algorithms, e.g., computer algebra systems, constraint solvers, model generators, and machine-oriented automated theorem provers. They can support the proof planning process by performing computations, detecting inconsistencies, suggesting instantiations of variables, or solving subproblems.

In general, ΩMEGA's proof planning can treat computations from external systems in two ways: as *hints* or as *proof steps*. The difference is that the soundness of hints is checked by the subsequent proof planning process, which either fails or succeeds for the given hint. To guarantee the soundness of proof steps, special procedures have to be provided, which transform the output of an external system into a subproof that ΩMEGA can check, i.e., special procedures that perform the expansion of such proof steps to ND. Technically, the interface of proof planning to external systems is realized by the LISP functions of methods and control rules. Methods can call external systems in their application conditions and outline computations;[7] similarly, control rules can employ external systems in the predicates of their IF-part.

Figure 4 and Figure 5 show the two methods COMPLEXESTIMATE-B and TELLCS-B whose application conditions comprise calls to external systems, respectively. Both methods are central for planning limit problems (see section 3.5).

COMPLEXESTIMATE-B is a method for estimating the magnitude of the absolute value of complex terms.[8] COMPLEXESTIMATE-B is applicable to tasks whose goal has the formula $|b| < \epsilon$ (corresponding to line L_9 in Figure 4) and that have supports with formula $|a| < \epsilon'$ (corresponding to line L_1 in Figure 4). In its application conditions COMPLEXESTIMATE-B uses

[7] Technically, calls of external systems in the expansion computations of methods are also possible. Currently, there is no method that performs such calls.

[8] COMPLEXESTIMATE-B essentially is a reconstruction (see [41]) of BLEDSOE'S limit heuristic that was used in a special-purpose program [8].

Method: COMPLEXESTIMATE-B	
declarations	variables: b, ϵ, a, ϵ', l, k, $a\sigma$, $k\sigma$, $l\sigma$, $b\sigma$, $\epsilon\sigma$, $\epsilon'\sigma$, $conjunct$, σ meta-variables: mv
parameters	
appl. conds.	*linearextract*(a, b, l, k, σ)
premises	L_1, $\oplus L_2$, $\oplus L_4$, $\oplus L_5$, $\oplus L_6$, $\oplus L_7$
conclusions	$\ominus L_9$
outline computations	$a\sigma :=$ *subst-apply*(σ, a) $k\sigma :=$ *subst-apply*(σ, k) $l\sigma :=$ *subst-apply*(σ, l) $b\sigma :=$ *subst-apply*(σ, b) $\epsilon\sigma :=$ *subst-apply*(σ, ϵ) $\epsilon'\sigma :=$ *subst-apply*(σ, ϵ') $conjunct :=$ *form-conjunction*(σ)
expansion computations	
proof schema	L_1. △ $\vdash \lvert a\rvert < \epsilon'$ $()$ L_2. △ $\vdash \epsilon'\sigma < \frac{\epsilon\sigma}{2*mv}$ $(Open)$ L_3. △ $\vdash \lvert a\sigma\rvert < \frac{\epsilon\sigma}{2*mv}$ $(< trans\ L_1\ L_2)$ L_4. △ $\vdash \lvert k\sigma\rvert \leq mv$ $(Open)$ L_5. △ $\vdash \lvert l\sigma\rvert < \frac{\epsilon\sigma}{2}$ $(Open)$ L_6. △ $\vdash 0 < mv$ $(Open)$ L_7. △ $\vdash conjunct$ $(Open)$ L_8. △ $\vdash b\sigma \doteq k\sigma * a\sigma + l\sigma$ (CAS) L_9. △ $\vdash \lvert b\rvert < \epsilon$ $(fix\ L_3\ L_4\ L_5\ L_6\ L_7\ L_8)$

Figure 4: The COMPLEXESTIMATE-B method.

the function *linearextract*. When applied to a and b *linearextract* employs the computer algebra system MAPLE [52] to compute suitable terms k and l such that $b = k * a + l$ holds. *linearextract* also computes a substitution σ such that $b\sigma = k\sigma * a\sigma + l\sigma$ holds (where $b\sigma, k\sigma, l\sigma$ result from b, k, l by the application of the substitution σ, respectively). Thereby, the substitution σ maps meta-variables in a, b to terms. COMPLEXESTIMATE-B is applicable only, if MAPLE provides k and l such that *linearextract* evaluates to true. If this is the case, the application of a corresponding action of the method reduces the original task to five tasks whose goals correspond to the lines L_2, L_4, L_5, L_6, L_7 in Figure 4. L_7 has the formula $conjunct$, which is computed from the substitution σ by the function *form-conjunction*. This formula is the conjunction of the mappings of the substitution σ. That is, if σ maps the meta-variables $mv_1, \ldots, mv_n$ to the terms $t_1, \ldots, t_n$, respectively, then $conjunct$ has the form $mv_1 \doteq t1 \wedge \ldots \wedge mv_n \doteq t_n$. If σ is empty, then $conjunct$ is simply $True$, the primitive truth. The justification fix for L_9 in the proof schema is only an abbreviation that stands for a sequence of about 20 tactic steps that comprises, in particular, an application of the triangle inequality. The application of MAPLE is reflected in line L_8 of the proof schema, which is justified by the tactic CAS. When this tactic is expanded, it employs the SAPPER [58] system to obtain a formal proof of the statement $b\sigma = k\sigma * a\sigma + l\sigma$ suggested by MAPLE.

For instance, when applied to a task with formula $|(f(c_x) - g(c_x)) - (l_1 - l_2)| < \epsilon$ and a support with formula $|f(mv_x) - l_1| < \epsilon'$ with a meta-variable mv_x, then *linearextract* succeeds and provides $k = 1$, $l = g(c_x) - l_2$, and a substitution σ that maps mv_x to c_x. The application of a corresponding action of ComplexEstimate-B reduces the given task to new tasks whose goals are $|1| \leq mv$, $\epsilon' < \frac{\epsilon}{2*mv}$, $|g(c_x) - L_2| < \frac{\epsilon}{2}$, $0 < mv$, and $mv_x \doteq c_x$.

Method: TellCS-B	
declarations	variables: a, b, rel
parameters	
appl. conds.	(1) *metavar-in*$(a) \vee$ *metavar-in*(b) (2) *test-CS*($\mathcal{C}o\mathcal{SIE}$,$rel(a,b)$)
premises	
conclusions	$\ominus L_1$
outline computations	
expansion computations	
proof schema	$L_1. \quad \triangle \quad \vdash rel_{o\nu\nu}(a_\nu, b_\nu) \quad (\mathit{ProveCS})$

Figure 5: The TellCS-B method.

The method TellCS-B realizes an interface to $\mathcal{C}o\mathcal{SIE}$ [47], a constraint solver for inequalities and equations over the field of real numbers. TellCS-B is applicable to tasks with formulas $rel(a, b)$ where rel is a binary predicate. Examples of matching predicates are, for instance, $<, \leq$. In its application conditions TellCS-B first tests whether a or b contain some meta-variables. If this is the case, $rel(a, b)$ is interpreted as a constraint on these meta-variables. TellCS-B applies then the function *test-CS* that connects to $\mathcal{C}o\mathcal{SIE}$ to test (1) whether $rel(a, b)$ is a syntactically valid constraint for $\mathcal{C}o\mathcal{SIE}$ (in particular, rel has to be $<, \leq, >, \geq, \doteq$, or $\neq$) and (2) whether $rel(a, b)$ is consistent with the current constraint store of $\mathcal{C}o\mathcal{SIE}$. If this is the case, TellCS-B is applicable and the corresponding action of TellCS-B contains in its **constraints** slot the constraint $rel(a, b)$. The introduction of the action closes the goal without producing further subtasks and passes $rel(a, b)$ as new constraint to $\mathcal{C}o\mathcal{SIE}$.

Figure 6 shows an action of the method TellCS-B. This action contains the constraint $0 < mv_D$, which is annotated with $\mathcal{C}o\mathcal{SIE}$ to indicate that the constraint has to be passed to $\mathcal{C}o\mathcal{SIE}$. The constraint results from the evaluation of the application condition *test-cs* of TellCS-B.

Action	
method	TellCS-B
task	$L_{10} \blacktriangleleft \{L_4, L_5\}$
premises	
conclusions	$\ominus\ L_{10}.\ L_4, L_5\ \vdash\ 0 < mv_D\ (Open)$
binding	$\{L_1 \rightarrow L_{10}, a \rightarrow 0, b \rightarrow mv_D, rel \rightarrow <\}$
constraints	$\{\mathcal{C}o\mathcal{SIE}$:$0 < mv_D\}$

Figure 6: An action with the TellCS-B method.

$\mathcal{C}o\mathcal{SIE}$ can provide instantiations of the constrained meta-variables that are consistent with the collected constraints. For instance, suppose during the proof planning process there are three tasks whose goals have the formulas $0 < mv_D$, $mv_D < \delta_1$, $mv_D < \delta_2$, which all contain the meta-variable mv_D. All three goals are closed by actions of TELLCS-B. Moreover, suppose there are also two supports with formulas $0 < \delta_1$ and $0 < \delta_2$, which are passed to $\mathcal{C}o\mathcal{SIE}$ by actions of the method TELLCS-F, which is the analogous of TELLCS-B to pass constraints in supports to $\mathcal{C}o\mathcal{SIE}$. From the resulting constraint store, $\mathcal{C}o\mathcal{SIE}$ can compute $min(\delta_1, \delta_2)$ as suitable instantiation for mv_D. Moreover, $\mathcal{C}o\mathcal{SIE}$ provides traces of its computations, which can be used to expand the applications of the actions of TELLCS-B.

Another method that establishes a connection to $\mathcal{C}o\mathcal{SIE}$ is ASKCS-B. Similar to TELLCS-B, this method is applicable to tasks whose goal formulas are of the form $rel(a, b)$. But whereas TELLCS-B demands that a or b contain some meta-variables, ASKCS-B covers the case that a and b contain no meta-variables. An application condition of ASKCS-B passes the formula to $\mathcal{C}o\mathcal{SIE}$ and asks $\mathcal{C}o\mathcal{SIE}$ whether the formula holds with respect to the constraints collected so far. If this is the case, then ASKCS-B closes the goal. Since $\mathcal{C}o\mathcal{SIE}$ can also handle formulas on concrete real numbers, for instance, $1 < 2$ or $0 \leq 0$, ASKCS-B can also close goals whose formulas are expressions on concrete real numbers.

Note that besides TELLCS-B and TELLCS-F also the methods $\forall$I-B and $\exists$E-F pass constraints to $\mathcal{C}o\mathcal{SIE}$. Actions of $\forall$I-B perform backward applications of the ND-rule $\forall_I$ by reducing a task with task formula $\forall x.P[x]$ to a new task with task formula $P[c]$, where the variable x is replaced by a constant c. For each meta-variable mv in $P[c]$ an action of $\forall$I-B also passes the *Eigenvariable constraint* $c!\not\in mv$ to $\mathcal{C}o\mathcal{SIE}$ that states that the instantiation for mv is not allowed to contain c. This constraint guarantees the adherence with the Eigenvariable conditions of the $\forall_I$ rule of the ND-calculus. Actions of the $\exists$E-F method perform a forward step with the $\exists_E$ rule. Similar to action of $\forall$I-B they pass Eigenvariable constraints to $\mathcal{C}o\mathcal{SIE}$ that demand the adherence of the Eigenvariable conditions of the $\exists_E$ rule.

3 Proof Planning with PLAN

PLAN is ΩMEGA's simple proof planner. It proceeds by successively computing and introducing actions into a proof plan under construction. Preceding the formal description of PLAN (see section 3.2), Table 1 shows the skeleton of PLAN's algorithm. Essentially, PLAN follows the precondition achievement paradigm (see section 2.1). First, it selects a task to work on. Then, it computes actions for this task and selects one action, which it introduces into the proof plan under construction. This results in new tasks on which PLAN continues. If PLAN fails to compute an action for a selected task, then it performs backtracking. Although actions can perform both, forward reasoning and backward reasoning, an action is always chosen with respect to a task in order to close or to reduce the gap between the goal and the supports of the task.[9] Some decisions in PLAN can be guided by control rules, for instance, the selection of the next task and the selection of the next action. Other decisions, however, are hard-coded into the system. For instance, PLAN employs backtracking if and only if it tackles a task, for which it fails to compute an action. Moreover, it employs external constraint solvers to obtain instantiations for meta-variables if and only if the agenda is empty and the $\mathcal{PDS}$ is closed.

[9] In the existing implementation PLAN can introduce a forward action with respect to several tasks simultaneously. This corresponds to the successive application of several actions to a single task, respectively. In order to simplify the formal discussion of PLAN we shall describe the action introduction only with respect to one task.

1. When the current agenda is empty and the current $\mathcal{PDS}$ is closed, then apply external constraint solvers to compute variable instantiations consistent with the collected constraints and terminate.
2. Select a task T from the agenda.
3. Compute and select an action A with respect to T.
4. If an action A could be computed for T, then introduce A. Goto step 1.
5. If no action A could be computed for T, then backtrack the action whose introduction created the task T. Goto step 1.

Table 1: Cycle of PLAN.

A node in the search space of PLAN is given by a set of tasks, i.e., an agenda. PLAN starts with the initial agenda. The next node in the search space is reached by the introduction of an action, which changes the agenda etc. A forward action creates a new task by changing the supports of a given task whereas a backward action replaces a task by some new tasks with new goals. The planning process stops as soon as a node in the search space is reached whose set of tasks is empty.

Proof planning does not suffer from the conjunctive goal problems of AI-planners that perform precondition achievement planning.[10] The derivation of a formula F in the subplan for a subgoal is not threatened or removed by the derivation of the negated formula $\neg F$ in the subplan for another subgoal. Hence, PLAN does not perform any threat resolution like demotion or promotion of actions. Moreover, since no re-ordering of introduced actions is performed, PLAN is a total-order planner that computes a sequence of actions.

PLAN's subprocedure for action deletion performs *dependency-directed backtracking* (e.g., see [59]). Instead of backtracking to the last decision point (so-called *chronological backtracking*), the idea of dependency-directed backtracking is to analyze which decisions along a search branch caused a failure. Then, decisions are removed and alternatives are tried based on the found dependencies, rather than the chronological order in which decisions were made. Since there is some ambiguity in the previous use of the term dependency-directed backtracking. We use the term as defined in [53] (p. 212): "*Sometimes, though, we have additional information that tells us which guess (along a search branch) caused the problem. We'd like to retract only that guess and the work that explicitly depended on it, leaving everything else that has happened in the meantime intact. This is exactly what dependency-directed backtracking does.*" Note that in this approach dependency-directed backtracking does not return to an already visited search state but can lead to a new state not visited before. In [24] the same approach is called *dynamic backtracking* because of the dynamic way in which the search is structured. In [28] the term dependency-directed backtracking refers to the approach that analyzes which decision caused a failure and to backtrack to this choice point. That is, all steps done after this decision are removed and an already visited search state is reached again.

[10] Given a conjunctive goal, it seems natural to try divide and conquer, but the subplans achieving the single subgoals may interfere and do not achieve the desired goals together. A famous example for this problem is the so-called "Sussman anomaly" problem in the blocks world. A detailed discussion of planning in the blocks world as well as the "Sussman anomaly" can be found in standard AI-textbooks, e.g., in [54].

Besides the information on the current planning state PLAN has also to maintain information on the search performed so far. In particular, it is necessary to store and make use of information on failing decisions in order to try alternatives instead. Search procedures that perform chronological backtracking often use search trees, which capture possible alternatives as well as made and failed decisions to store information on the traversed search space (e.g., see [1]). Since PLAN performs dependency-directed backtracking we decided for a different approach. PLAN maintains a so-called *history*. A history is a sequence of *manipulation records*. Figure 7 shows the skeletons of the two manipulation records, the *action-introduction record* and the *action-deletion record*, of PLAN.

Action-Introduction:	
agenda	
introduced-action	
alternatives	
new-tasks	

Action-Deletion:	
agenda	
deleted-action	

Figure 7: Manipulation records in PLAN.

The slot **agenda** captures the context in which the manipulation was done (i.e., the agenda before the manipulation), the slots **introduced-action** and **deleted-action** capture the performed manipulation (i.e., the introduced or deleted action), the slot **alternatives** captures alternative actions available as the introduced action was chosen, and the slot **new-tasks** captures the new tasks created by the application of the chosen action. PLAN records each action introduction or deletion with a corresponding entry in the history. It makes direct use of this information, when selecting the next action: it does not choose again an action that was already deleted (see section 3.4). Since PLAN does not return to a particular search state it does not make direct use of the stored alternative actions. However, the information of the history is available to the control rules, which can reason on backtracked steps and possible alternative actions.[11]

In the remainder of this section, we give a detailed description of PLAN. First, we give some formal definitions that culminate in a definition of proof plans and solution proof plans. Then, the subsequent sections give detailed descriptions of PLAN's main algorithm and its subalgorithms for action computation and deletion. As conclusion of the section, we discuss a sample application of PLAN.

Notation In the remainder of the report, the following symbols (maybe labeled with some subscripts or superscripts) are associated with the following objects:

$\vec{A}$	denotes a sequence of actions,
$\mathcal{P}$	denotes a $\mathcal{PDS}$,
Â	denotes an agenda,
$\vec{H}$	denotes a history.

3.1 Formal Definition of Proof Plans in PLAN

The aim of this section is to give a formal description of proof plans. We start with definitions of a proof planning problem, an initial $\mathcal{PDS}$ of a proof planning problem, and an initial agenda of a

[11] We are currently extending manipulation records to capture also information on the reasons that support a certain decision.

proof planning problem.

Definition A *proof planning problem* is a quadruple $(Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{M}, \mathcal{C})$ where Thm and $Ass_1, \ldots, Ass_n$ are formulas in ΩMEGA's higher-order language, $\mathcal{M}$ is a set of methods, and $\mathcal{C}$ is a set of control rules. Thm is also called the *theorem of the proof planning problem* whereas $Ass_1, \ldots, Ass_n$ are called the *assumptions of the proof planning problem.* □

Definition Let $(Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{M}, \mathcal{C})$ be a proof planning problem. The *initial* $\mathcal{PDS}$ of this proof planning problem is the $\mathcal{PDS}$ that consists of an open line L_{Thm} with formula Thm and the lines L_{Ass_i} with formula Ass_i and the hypothesis justification Hyp, respectively. The *initial agenda* of the proof planning problem is the agenda that consists of the task $L_{Thm} \blacktriangleleft \{L_{Ass_1}, \ldots, L_{Ass_n}\}$. The task $L_{Thm} \blacktriangleleft \{L_{Ass_1}, \ldots, L_{Ass_n}\}$ is also called the *initial task* of the proof planning problem. □

Next, we define, when an action is applicable with respect to a $\mathcal{PDS}$. Informally speaking, this is the case, when the given lines of the action are in the $\mathcal{PDS}$. Afterwards, we introduce the action introduction function Φ, which describes the operational semantics of an action when it is applied to an agenda, a $\mathcal{PDS}$, and a sequence of actions (i.e., Φ defines a transition relation between triples of agendas, $\mathcal{PDS}$s, and sequences of actions).

Definition Let $\mathcal{P}$ be a $\mathcal{PDS}$ and A_{add} an action. Moreover, let $\mathcal{L}$ be the set of proof lines of $\mathcal{P}$ and let $\ominus Concs$ be the $\ominus$ conclusions, $\ominus Prems$ the $\ominus$ premises, and $BPrems$ the blank premises of A_{add}.

A_{add} is *applicable* with respect to $\mathcal{P}$ if

- $(\ominus Concs \cup \ominus Prems \cup BPrems)$ is a subset of $\mathcal{L}$.

□

Definition The *action introduction function* Φ is a partial function that maps a sequence of actions, an agenda, a $\mathcal{PDS}$, and an action into a sequence of actions, an agenda, and a $\mathcal{PDS}$, i.e.,

$$\Phi : \vec{A} \times \hat{A} \times \mathcal{P} \times A_{add} \mapsto \vec{A}\text{'} \times \hat{A}\text{'} \times \mathcal{P}\text{'}.$$

Let A_{add} be an action that is applicable with respect to the $\mathcal{PDS}$ $\mathcal{P}$. Let $\oplus Concs$ be the $\oplus$ conclusions, $\ominus Concs$ the $\ominus$ conclusions, $\oplus Prems$ the $\oplus$ premises, $\ominus Prems$ the $\ominus$ premises, and $BPrems$ the blank premises of A_{add}. Moreover, let $T = L_{open} \blacktriangleleft \mathcal{SUPPS}_{L_{open}}$ be the task of A_{add}.

$Prems$:=$\oplus Prems \cup \ominus Prems \cup BPrems$,
$Concs$:=$\oplus Concs \cup \ominus Concs$
$New\text{-}Lines$:=$\oplus Concs \cup \oplus Prems$
$New\text{-}Supps$:=$(\mathcal{SUPPS}_{L_{open}} \cup \oplus Concs) - \ominus Prems$.
$New\text{-}Tasks$:=$[L \blacktriangleleft New\text{-}Supps \mid L \in \oplus Prems]$.

If $\vec{A}$ is a sequence of actions and $\hat{A}$ is an agenda that contains the task T of A_{add}, then the result $(\vec{A}\text{'}, \hat{A}\text{'}, \mathcal{P}\text{'})$ of $\Phi(\vec{A}, \hat{A}, \mathcal{P}, A_{add})$ is defined by:

- $\vec{A}$':= $\vec{A} \cup [A_{add}]$.
- Â':= $\begin{cases} \mathit{New\text{-}Tasks} \cup (\hat{A} - [T]) & \text{if } L_{open} \in \ominus \mathit{Concs}, \\ [L_{open} \blacktriangleleft \mathit{New\text{-}Supps}] \cup \mathit{New\text{-}Tasks} \cup (\hat{A} - [T]) & \text{else.} \end{cases}$
- $\mathcal{P}$' results from $\mathcal{P}$ by
  1. adding the proof lines $\mathit{New\text{-}Lines}$, respectively, and
  2. justifying the proof lines $\ominus \mathit{Concs}$ and $\oplus \mathit{Concs}$ with the justification $(M\ \mathit{Prems})$, respectively, where M is the method of A_{add}.

□

The recursive extension Φ is called $\vec{\Phi}$. $\vec{\Phi}$ introduces a whole sequence of actions (the arrow of $\vec{\Phi}$ indicates that this function introduces a sequence of actions $\vec{A}_{add}$).

Definition The *recursive action introduction function* $\vec{\Phi}$ is a partial function that maps a sequence of actions, an agenda, a $\mathcal{PDS}$, and a sequence of actions into a sequence of actions, an agenda, and a $\mathcal{PDS}$, i.e.,

$$\vec{\Phi} : \vec{A} \times \hat{A} \times \mathcal{P} \times \vec{A}_{add} \mapsto \vec{A}' \times \hat{A}' \times \mathcal{P}'.$$

$\vec{\Phi}$ is recursively defined as follows:
Let $\vec{A}$ be a sequence of actions, Â an agenda, $\mathcal{P}$ a $\mathcal{PDS}$, and $\vec{A}_{add}$ a sequence of actions.

1. If $\vec{A}_{add}$ is empty then $\vec{\Phi}(\vec{A}, \hat{A}, \mathcal{P}, \vec{A}_{add}) = (\vec{A}, \hat{A}, \mathcal{P})$.
2. Otherwise let $A_{add} := \mathit{first}(\vec{A}_{add})$ and $\vec{A}'_{add} := \mathit{rest}(\vec{A}_{add})$. If A_{add} is applicable with respect to $\mathcal{P}$, and if Â contains the task of A_{add}, then
$$\vec{\Phi}(\vec{A}, \hat{A}, \mathcal{P}, \vec{A}_{add}) = \vec{\Phi}(\Phi(\vec{A}, \hat{A}, \mathcal{P}, A_{add}), \vec{A}'_{add}).$$

□

With the function $\vec{\Phi}$ we can now define proof plans and solution proof plans.

Definition
Let $(\mathit{Thm}, \{\mathit{Ass}_1, \ldots, \mathit{Ass}_n\}, \mathcal{M}, \mathcal{C})$ be a proof planning problem, $\mathcal{P}_{init}$ the initial $\mathcal{PDS}$ of this problem, and Â$_{init}$ its initial agenda.

A *proof plan* for the proof planning problem is a triple $\mathsf{PP} = (\vec{A}, \hat{A}, \mathcal{P})$ with a sequence of actions $\vec{A}$, an agenda Â, and a $\mathcal{PDS}$ $\mathcal{P}$ such that:

1. the methods of each action of $\vec{A}$ are in $\mathcal{M}$,
2. $(\vec{A}, \hat{A}, \mathcal{P}) = \vec{\Phi}([], \hat{A}_{init}, \mathcal{P}_{init}, \vec{A})$,

A *solution proof plan* for the proof planning problem is a sequence of actions $\vec{A}$ such that $\vec{\Phi}([], \hat{A}_{init}, \mathcal{P}_{init}, \vec{A})$ has an empty agenda and a closed $\mathcal{PDS}$. □

Because of this definition, we can also say that Φ maps a proof plan and an action into a proof plan and that $\vec{\Phi}$ maps a proof plan and a sequence of actions into a proof plan.

3.2 The PLAN Algorithm

Figure 8 gives a pseudo-code description of the PLAN algorithm. PLAN obtains as input a proof plan $\mathsf{PP} = (\vec{A}, \hat{\mathrm{A}}, \mathcal{P})$, a history $\vec{H}$, a list of methods $\mathcal{M}$, and a list of control rules $\mathcal{C}$.[12] PLAN generates a sequence of pairs of proof plans PP and histories $\vec{H}$. The user of ΩMEGA can start PLAN with the initial $\mathcal{PDS}$, the initial agenda, and the set of methods and control rules of a proof planning problem. In order to reach the next proof plan and the next history PLAN performs a cycle of termination check, task selection, action selection and action introduction or action deletion. It terminates when either the agenda of the current proof plan is empty (see step 1 in Figure 8) or when there are neither further actions to be introduced nor actions to be removed (see step 5 in Figure 8). In the former case PLAN was successful and returns the proof plan and the constructed closed $\mathcal{PDS}$. In the latter case, PLAN did traverse the complete search space without finding a proof plan and returns **fail**.

If the current agenda is not empty, then PLAN first selects the next task to tackle (step 2 in Figure 8). To do so, PLAN employs the function *evalcrules-tasks*. *evalcrules-tasks* evaluates the control rules $\mathcal{C}$ of the kind ‘Tasks’ on the tasks list of the current agenda and returns a (possibly) changed alternative list.[13] Then, PLAN picks the fist element of the resulting list as current task.

Next, PLAN employs the subalgorithm **CHOOSEACTION** to compute an action (step 3 in Figure 8). **CHOOSEACTION** is applied to the current task, the methods $\mathcal{M}$, and the control rules $\mathcal{C}$. It tries to compute admissible actions and – if successful – it selects one action and returns it. Since **CHOOSEACTION** is a complex algorithm we shall discuss it in detail in section 3.4.

If **CHOOSEACTION** returns an action, then PLAN introduces the action (step 4 in Figure 8). It creates a new proof plan by applying the action introduction function Φ to the current proof plan and the chosen action. Moreover, it creates a new history by adding a new action-introduction record entry to the history. PLAN uses the function *extract-constraints* to access the constraints of an action. When the action contains constraints for the connected external constraint solvers, then PLAN employs the function *pass-constraints*, which passes the constraints to the respective external system. PLAN does not check whether the new constraints are accepted by the respective external system. Rather, it assumes that corresponding consistency checks are performed by **CHOOSEACTION** as part of the evaluation of the application conditions of a method, when an action is computed.

When **CHOOSEACTION** fails to provide an action, then PLAN tries to delete actions in the current proof plan (step 5 in Figure 8). If the current sequence of actions is empty, then this is obviously not possible. When there are no more actions that can be introduced and the current sequence of actions is empty, then PLAN did traverse the complete search space (complete wrt. to the methods $\mathcal{M}$ and the control rules $\mathcal{C}$) without finding a solution proof plan. In this case, PLAN terminates and returns **fail**. If there are actions that can be deleted, then PLAN employs the function *find-introducing-action* to determine the action whose introduction created the task T for which no action can be computed. The information about which action introduction did introduce which task can be found in the history in the action-introduction entries. Then, PLAN employs

[12] Both methods $\mathcal{M}$ and control rules $\mathcal{C}$ are lists and not sets since the order in these lists are relevant. The order in $\mathcal{M}$ gives a default order in which the methods are tried, when no control rules fire and determine a different order (see section 3.4). The order in $\mathcal{C}$ determines the order in which the control rules are evaluated.

[13] Although we do not explicitly provide the current proof plan and the current history as arguments for *evalcrules-tasks*, the predicates in the IF-part of the evaluated control rules can make use of this status information. This holds for all kinds of control rules, not only for the control rules of kind ‘Tasks’ evaluated here.

Input: (1) a proof plan PP $= (\vec{A}, \hat{A}, \mathcal{P})$ with a sequence of actions $\vec{A}$, an agenda $\hat{A}$, and a $\mathcal{PDS}$ $\mathcal{P}$, (2) a history $\vec{H}$, (3) a list of methods $\mathcal{M}$, (4) a list of control rules $\mathcal{C}$.

Output: Either a solution proof plan and a closed $\mathcal{PDS}$ or **fail**.

Algorithm: PLAN$((\vec{A}, \hat{A}, \mathcal{P}),\vec{H},\mathcal{M},\mathcal{C})$

```
1. Termination
   If $\hat{A}$ is empty, then terminate and return employ-CS$(\vec{A},\mathcal{P})$.

2. Task Selection
   Let current task $T$:= first(evalcrules-tasks$(\hat{A},\mathcal{C})$)
   where $T$ is the pair $L_{open} \blacktriangleleft \mathcal{SUPPS}_{L_{open}}$.

3. Action Selection
   Let $(A_{add},\mathcal{A})$:=CHOOSEACTION$(T,\vec{H},\mathcal{M},\mathcal{C})$
   where $A_{add}$ is an action and $\mathcal{A}$ is a set of alternative actions.

4. Action Introduction
   If $A_{add}$ is given
     then
       $(\vec{A}',\hat{A}',\mathcal{P}')$:=$\Phi(\vec{A}, \hat{A}, \mathcal{P}, A_{add})$.
       $\vec{H}'$:=add-action-intro-record$(\vec{H},\hat{A},A_{add},\mathcal{A})$.
       If extract-constraints$(A_{add}) \neq \emptyset$
         then
           pass-constraints(extract-constraints$(A_{add})$).
       PLAN$((\vec{A}',\hat{A}',\mathcal{P}'),\vec{H}',\mathcal{M},\mathcal{C})$.

5. Action Deletion
   If $A_{add}$ is not given
     then
       If $\vec{A}$ is empty
         then
           Terminate and return fail.
         else
           Let $A_{reason}$:=find-introducing-action$(T,\vec{H})$.
           $((\vec{A}',\hat{A}',\mathcal{P}'),\vec{H}')$:=BACKTRACK$((\vec{A},\hat{A},\mathcal{P}),\vec{H},[A_{reason}])$.
           PLAN$((\vec{A}',\hat{A}',\mathcal{P}'),\vec{H}',\mathcal{M},\mathcal{C})$.
```

Figure 8: The PLAN algorithm.

the subalgorithm **BACKTRACK** to perform the deletion of the selected action and all further actions that explicitly depend on it. **BACKTRACK** is applied to the current proof plan, the current history, and a list with the action to be deleted as only element. It returns a changed proof plan and a changed history. Since **BACKTRACK** is a complex algorithm we shall discuss it in detail in the next section.

When the agenda is empty, then the introduction of actions stops and PLAN applies the function *employ-CS* to the computed action sequence and the constructed $\mathcal{PDS}$ (step 1 in Figure 8). This function employs the external constraint solvers to compute instantiations for the meta-variables. Then, it substitutes all occurrences of the meta-variables in proof lines of the $\mathcal{PDS}$ and the actions by their instantiations, respectively. It returns the resulting action sequence and the instantiated $\mathcal{PDS}$, which are then the output of PLAN.

Although proof planning actions are complex actions in the sense of HTN-planning, the expansion of actions is not performed within PLAN. Rather, there are separate procedures in ΩMEGA for the expansion of actions. When an expansion fails to produce a calculus-level proof and results in new open lines, then PLAN can be re-invoked on the new tasks.

3.3 Deletion of Actions

Before we describe the **BACKTRACK** algorithm, we shall introduce the notion of dependency among actions and when an action is deletable. When an action is introduced into a proof plan, then it modifies the elements of the proof plan. Other actions introduced later on may depend on these modifications. More concretely, when the new lines introduced by an action are used as given lines by other actions introduced later on, then these actions depend on the preceding action. Afterwards, we define the function for the deletion of an action from a proof plan. Since action deletion is conceptually the inverse operation of action introduction we call this function Φ^{-1} although technically Φ^{-1} is not the inverse function of Φ.

Definition Let $\vec{A}$ be a sequence of actions with

$$\vec{A}=[A_1,\ldots,A_{i-1},A_i,A_{i+1},\ldots,A_n].$$

Let A_i be an action with the $\oplus$ conclusions $\oplus Concs$, and the $\oplus$ premises $\oplus Prems$. An action $A_j \in \{A_{i+1},\ldots,A_n\}$ depends on A_i, if A_j is an action whose sets of conclusions or premises contains a proof line of $\oplus Concs$ or $\oplus Prems$ (which are the new proof lines introduced by A_i). □

Definition Let $\vec{A}$ be a sequence of actions with

$$\vec{A}=[A_1,\ldots,A_{i-1},A_{del},A_{i+1},\ldots,A_n].$$

A_{del} is *deletable* with respect to $\vec{A}$, if the set of actions in $\vec{A}$ that depend on A_{del} is empty. □

In the following definition of the function Φ^{-1} we describe the modifications of the sequence of actions, the agenda, and the $\mathcal{PDS}$ caused by the deletion of an action. Although the notion of deletability of an action is defined only with respect to a sequence of actions, we demand in the definition of Φ^{-1} that the agenda and the $\mathcal{PDS}$ are not arbitrary ones, but created by this sequence of actions (in particular, by the action that should be deleted). The described modifications cannot be performed with respect to an arbitrary $\mathcal{PDS}$ or an arbitrary agenda.

Definition The *action deletion function* Φ^{-1} is a partial function that maps a sequence of actions, an agenda, a $\mathcal{PDS}$, and an action into a sequence of actions, an agenda, and a $\mathcal{PDS}$, i.e.,

$$\Phi^{-1} : \vec{A} \times \hat{\mathrm{A}} \times \mathcal{P} \times A_{del} \mapsto \vec{A}\text{'} \times \hat{\mathrm{A}}\text{'} \times \mathcal{P}\text{'}.$$

Let A_{del} be a deletable action in $\vec{A}$. Let $\oplus Concs$ be the $\oplus$ conclusions, $\ominus Concs$ the $\ominus$ conclusions, $\oplus Prems$ the $\oplus$ premises, $\ominus Prems$ the $\ominus$ premises, and $BPrems$ the blank premises of A_{del}. Moreover, let $T = L \blacktriangleleft \mathcal{SUPPS}_L$ be the task of A_{del}.
$Lines\text{-}To\text{-}Remove$:=$\oplus Concs \cup \oplus Prems$.
$Tasks\text{-}To\text{-}Remove$:=$[L \blacktriangleleft \mathcal{SUPPS}_L \in$ Â $\mid L \ \in \oplus Prems]$.
$New\text{-}Tasks$:=$[T]$.

If Â is an agenda and $\mathcal{P}$ is a $\mathcal{PDS}$ that results from the introduction of $\vec{A}$ (to some agenda and some $\mathcal{PDS}$), then the result $(\vec{A}$', Â', $\mathcal{P}$') of $\Phi^{-1}(\vec{A}$, Â, $\mathcal{P}, A_{del})$ is defined by:

- $\vec{A}$':= $\vec{A} - [A_{del}]$.
- Â':= $New\text{-}Tasks \cup$ (Â $- Tasks\text{-}To\text{-}Remove$).
- $\mathcal{P}$' results from $\mathcal{P}$ by
  1. removing the lines $Lines\text{-}To\text{-}Remove$ and
  2. justifying the proof lines $\ominus Concs$ with $Open$, respectively.

□

A pseudo-code description of the algorithm **BACKTRACK** is given in Figure 9. **BACKTRACK** is applied to a proof plan PP $= (\vec{A}$, Â, $\mathcal{P})$, a history $\vec{H}$, and a list of actions $\vec{A}_{del}$ that have to be deleted. **BACKTRACK** generates a sequence of pairs of proof plans PP and histories $\vec{H}$ by deleting successively the actions in $\vec{A}_{del}$. If an action in $\vec{A}_{del}$ is not deletable, then it is necessary to delete further actions. **BACKTRACK** returns the proof plan and the history that result from the deletion of all necessary actions.

The first step in **BACKTRACK** is a check whether the list of actions that should be deleted is empty. If this is the case, **BACKTRACK** terminates and returns the current proof plan and the current history. Otherwise, it selects the first action A_{del} from the list (step 2 in Figure 9). If A_{del} is deletable, **BACKTRACK** deletes it from the current proof plan by employing Φ^{-1} and adds a new action-deletion entry to the history (step 3 in Figure 9). When A_{del} contains constraints, then **BACKTRACK** employs the function *delete-constraints*, which tells the respective constraint solvers to delete these constraints since they are not longer existing. Afterwards, **BACKTRACK** is applied to the changed proof plan, the changed history, and the remaining actions to be deleted.

If the action A_{del} is not deletable (step 4 in Figure 9), then **BACKTRACK** calls the function *dependent-actions* to compute the actions that depend from A_{del} and that have to be deleted in order to make A_{del} deletable. **BACKTRACK** is then recursively applied to the current proof plan, the current history, and the concatenation of the actions computed by *dependent-actions* and the current actions that have to be deleted.

As example for a situation, where an action is not deletable because other actions depend on it, consider the following situation. PLAN introduces an action A that reduces a task with goal L to two new tasks with goals L_1 and L_2. Next, PLAN applies the action A_1 to close L_1. Afterwards, PLAN fails to apply an action to the task with goal L_2 and employs **BACKTRACK** to remove the action A that introduced L_2. However, the deletion of A would not only remove the line L_2 but also the line L_1 with respect to which action A_1 was introduced. Hence, before A can be deleted the action A_1 has to be deleted.

Input: (1) a proof plan PP $= (\vec{A}, \hat{\mathrm{A}}, \mathcal{P})$ with a sequence of actions $\vec{A}$, an agenda $\hat{\mathrm{A}}$, and a $\mathcal{PDS}$ $\mathcal{P}$, (2) a history $\vec{H}$, (3) a sequence of actions $\vec{A}_{del}$.

Output: A proof plan PP' $= (\vec{A}', \hat{\mathrm{A}}', \mathcal{P}')$ and a history $\vec{H}'$.

```
Algorithm: Backtrack($(\vec{A}, \hat{A}, \mathcal{P}),\vec{H},\vec{A}_{del}$)

  1. Termination
     If $\vec{A}_{del}$ is empty, then terminate and return $((\vec{A}, \hat{A}, \mathcal{P}), \vec{H})$.

  2. Pick Action
     Let $A_{del}$:=first($\vec{A}_{del}$).

  3. Action Deletion
     If $A_{del}$ is deletable wrt. $\vec{A}$
       then
         $(\vec{A}', \hat{A}', \mathcal{P}') := \Phi^{-1}(\vec{A}, \hat{A}, \mathcal{P}, A_{del})$.
         $\vec{H}'$:=add-action-del-record($\vec{H}, \hat{A}, A_{del}$).

         If extract-constraints($A_{del}$) $\neq \emptyset$
           then
             delete-constraints(extract-constraints($A_{del}$)).

         BACKTRACK($(\vec{A}', \hat{A}', \mathcal{P}'),\vec{H}'$,rest($\vec{A}_{del}$)).

  4. Deletion Expansion
     If $A_{del}$ is not deletable wrt. $\vec{A}$
       then
         $\vec{A}_{del}^{new}$:=dependent-actions($A_{del}, \vec{A}$).
         BACKTRACK($(\vec{A}, \hat{A}, \mathcal{P}),\vec{H},\vec{A}_{del}^{new} \cup \vec{A}_{del}$).
```

Figure 9: The **BACKTRACK** algorithm.

3.4 Action Computation and Selection

CHOOSEACTION is the subalgorithm of PLAN that computes alternative lists of actions and selects one of them. Figure 10 shows a pseudo-code description of the algorithm. **CHOOSEACTION** is applied to a task, the current history, and the lists of methods $\mathcal{M}$ and control rules $\mathcal{C}$. If successful, **CHOOSEACTION** returns a selected action and a set of alternative actions (see step 7 in Figure 10), otherwise it returns **fail** (see step 2 in Figure 10).

Input: (1) a task T, (2) a history $\vec{H}$, (3) a list of methods $\mathcal{M}$, (4) a list of control rules $\mathcal{C}$.

Output: Either a pair of an action and a list of actions or **fail**.

```
Algorithm: ChooseAction($T$,$\vec{H}$,M,C)
Let $T$=$L_{open} \blacktriangleleft \mathcal{SUPPS}_{L_{open}}$.

  1. Order Methods
     $Methods$:= evalcrules-methods($\mathcal{M}$,$\mathcal{C}$,$T$).

  2. Select Method
     If $Methods$ empty
       then
          Terminate and return fail.
       else
          $M$:=first($Methods$).
          $Actions$:=initial-action-set($T$, $M$).

  3. Match Goal
     Let $\ominus Concs$ be the $\ominus$ conclusions of $M$.
     $Actions$:=match-goal($L_{open}$,$\ominus Concs$,$Actions$).
     If $Actions$ empty, then $Methods$:=rest($Methods$), goto 2.

  4. Select and Match Supports and Parameters
     Let $\ominus Prems$ and $BPrems$ be the $\ominus$ premises and blank premises of $M$. Let
     $Params$ be the parameter variables of $M$.
     $Supps$+$Params$:=evalcrules-s+p($\mathcal{SUPPS}_{L_{open}}$,$\mathcal{C}$,$T$,$M$,$Actions$).
     $Actions$:= match-s+p($Supps$+$Params$,$\ominus Prems \cup BPrems$,
                                                   $Params$,$Actions$).
     If $Actions$ empty, then $Methods$:=rest($Methods$), goto 2.

  5. Evaluate Application Conditions
     $Actions$:=eval-appl-conds($Actions$,$M$).
     If $Actions$ empty, then $Methods$:=rest($Methods$), goto 2.

  6. Outline Computations
     eval-outline-computations($Actions$).
     complete-outline($Actions$).

  7. Select an Action
     $Actions$:=remove-backtracked($Actions$,$\vec{H}$).
     $Actions$:=evalcrules-actions($Actions$,$\mathcal{C}$).
     If $Actions = \emptyset$
       then
          $Methods$:=rest($Methods$), goto 2.
       else
          Terminate and return (first($Actions$),rest($Actions$)).
```

Figure 10: The **ChooseAction** algorithm.

ChooseAction computes actions successively. It starts with an under-specified, initial action that contains only a chosen method and the given task. Then, it successively matches lines of the method with the goal and the supports of the task as well as variables specified in the declarations of the method with terms, positions, etc. The substitutions of these matchings refine successively the binding of the action such that more and more specified actions are created. In order to check whether a particular action of a method is valid, **ChooseAction** evaluates the application conditions of the method with respect to the binding of the action. Afterwards, it completes the binding of the actions by conducting the outline computations and by computing the new lines. Finally, it selects one action among the resulting fully specified actions.

In the following, we explain **ChooseAction** with the example 2.1 of section 2.4. We apply **ChooseAction** to the task $L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$, an empty history, a list of methods that contains =Subst-B, and a list of control rules that contains the control rule `supps+params-=Subst` whose impact is explained below.

The first step in **ChooseAction** is the re-ordering of the alternative list of methods. This is done by the function *evalcrules-methods*, which obtains as input $\mathcal{M}$, $\mathcal{C}$ and the given task. *evalcrules-methods* evaluates the control rules in $\mathcal{C}$ of kind 'Methods' on $\mathcal{M}$ and returns a (possibly) changed list of alternative methods. From this list **ChooseAction** picks the first one (step 2 in Figure 10) and employs the function *initial-action-set* to create the initial set of actions that consists of one action whose premises, conclusions, bindings, and constraints are empty, whose method is the chosen method, and whose task is the given task.

For our example, we assume that *evalcrules-methods* returns the list [=Subst-B, ...]. Then, **ChooseAction** chooses =Subst-B as method and produces an initial set of actions that contains only the following action:

Action	
method	=Subst-B
task	$L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$
premises	
conclusions	
binding	
constraints	

The next step (step 3 in Figure 10) in **ChooseAction** matches the goal with the $\ominus$ conclusions of the selected method. To do so, **ChooseAction** employs the function *match-goal*. This function is applied to the goal, the $\ominus$ conclusions of the selected method, and the set of actions computed so far. Its computations and its output depend on the existence of $\ominus$ conclusions in the chosen method. If the method has no $\ominus$ conclusions (i.e., a forward method), then *match-goal* simply returns the list of actions it obtained as input. If the method has $\ominus$ conclusions (i.e., a backward method), then *match-goal* matches the goal with the $\ominus$ conclusions, respectively. For each successful matching it creates a new action whose binding contains the substitution resulting from the matching and whose conclusions contain the goal annotated with $\ominus$. Finally, *match-goal* returns the set of all new actions.

In our example the matching of the goal L_{Thm} with the $\ominus$ conclusions of =Subst-B results in a substitution with two elements: $L_3 \mapsto L_{Thm}$ and $f \mapsto even(a + b)$. Thus, *match-goal* returns an actions set that contains only the following action:

Action	
method	=Subst-B
task	$L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$
premises	
conclusions	$\ominus\ L_{Thm}.\ L_{Ass_1}, L_{Ass_2} \vdash\ even(a+b)\ (Open)$
binding	$\{L_3 \mapsto L_{Thm}, f \mapsto even(a+b)\}$
constraints	

Next, **ChooseAction** chooses supports and parameters and matches them with $\ominus$ and blank premises and the parameter variables of the selected method (step 4 in Figure 10). This results in further substitutions, which refine the actions computed so far. First, **ChooseAction** evaluates the control rules of the kind 'Supps+Params'. This is done by the function *evalcrules-s+p*, which is applied to the supports of the goal, the control rules $\mathcal{C}$, the task, the current method, and the actions computed so far. Control rules of the kind 'Supps+Params' do not only reorder and manipulate the support lines but they return a new type of elements, namely pairs of support lines and parameter instantiations. Thus, the parameter selection is not an isolated decision but is combined with the selection of support lines.[14] Then, **ChooseAction** employs the function *match-s+p*. *match-s+p* obtains as input the pairs of support lines and parameter instantiations, the $\ominus$ and blank premises of the selected method, and the set of actions computed so far. With respect to each action computed so far (i.e., depending on the binding of an action computed so far) *match-s+p* matches the support lines and parameters pairs with the $\ominus$ and blank premises and the parameter variables of the method, respectively. For each successful matching it creates a new action whose binding is extended with the substitution resulting from the matching and whose premises comprise the matched support lines. Finally, *match-s+p* returns the set of new actions.

In our example, the control rule `supps+params-=Subst` fires and returns the two support lines and parameter instantiation pairs $(\{L_{Ass_1}\}, <1\ 1>)$ and $(\{L_{Ass_2}\}, <1\ 2>)$, where $<1\ 1>$ is the parameter position of the a in the formula $even(a+b)$ of the goal L_{Thm} and $<1\ 2>$ is the parameter position of the b.[15] For both pairs and with respect to the only action computed so far, *match-s+p* succeeds to match the premise L_1 and the parameter pos of =Subst-B with the content of the pairs, respectively. It returns a set of actions that contains the following two elements:

Action	
method	=Subst-B
task	$L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$
premises	$L_{Ass_1}.\ L_{Ass_1} \vdash\ a \doteq c\ (Hyp)$
conclusions	$\ominus\ L_{Thm}.\ L_{Ass_1}, L_{Ass_2} \vdash\ even(a+b)\ (Open)$
binding	$\{L_3 \mapsto L_{Thm}, L_1 \to L_{Ass_1}, f \mapsto even(a+b), \alpha \to \nu, t \to a, t' \to c, pos \to <1\ 1>\}$
constraints	

[14] We decided for this combined approach since typically the parameter selection is directly related to the support line selection.

[15] The control rule `supps+params-=Subst` fires if the current method is =Subst-B and if there are some support lines that are equations such that one side of the equations equals a subterm in the formula of the goal. If `supps+params-=Subst` finds such a support line it returns a pair consisting of the support line and the respective subterm position in the formula of the goal.

Action	
method	=Subst-B
task	$L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$
premises	$L_{Ass_2}.\ L_{Ass_2} \ \vdash\ b \doteq c\ (Hyp)$
conclusions	$\ominus\ L_{Thm}.\ L_{Ass_1}, L_{Ass_2} \ \vdash\ even(a+b)\ (Open)$
binding	$\{L_3 \mapsto L_{Thm}, L_1 \to L_{Ass_2}, f \mapsto even(a+b), \alpha \to \nu,$ $t \to b, t' \to c, pos \to < 1\ 2 >\}$
constraints	

The first action results from matching L_1 and pos with L_{Ass_1} and $< 1\ 1 >$, respectively, whereas the second action results from matching L_1 and pos with L_{Ass_2} and $< 1\ 2 >$, respectively.

In the next step (step 5 in Figure 10), **ChooseAction** evaluates the application conditions of the selected method. The evaluation of the application conditions is performed by the function *eval-appl-conds*, which obtains as input the actions computed so far and the selected method. For each given action *eval-appl-conds* evaluates the application conditions of the method with respect to the binding of the action. The evaluation of application conditions can create further substitutions, which are then added to the binding of the action. Moreover, the evaluation can create constraints for external constraint solvers, which are then added as constraints of the action. Each action for which the evaluation fails is rejected. *eval-appl-conds* returns the set of all actions for which the evaluation succeeds.

In our example, the application conditions of =Subst-B evaluate to **true** for both actions computed so far. Since no constraint results from the evaluation of the application conditions the constraints of both actions are set to the empty set.

Next, **ChooseAction** completes the actions by conducting the outline computations of the selected method and by computing the new outline lines (i.e., $\oplus$ premises and conclusions) (see step 6 in Figure 10). This is done by the functions *eval-outline-computations* and *complete-outline*, which both are applied to the set of actions computed so far. Both functions do not change the set of actions but they refine the actions already in the set. *eval-outline-computations* evaluates the outline computations for each action and adds the resulting substitutions to the binding of the action. Similarly, *complete-outline* computes the missing outline lines for each action and adds the corresponding substitutions to the binding of the action. New outline lines are justified as follows: $\oplus$ premises are justified with $Open$ whereas new $\oplus$ conclusions are justified by an application of the selected method to the premises of the action.

For our example, *eval-outline-computations* and *complete-outline* complete the actions computed so far as follows:

Action	
method	=Subst-B
task	$L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$
premises	$\oplus\ L_{Thm'}.\ L_{Ass_1}, L_{Ass_2} \ \vdash\ even(c+b)\ (Open)$ $L_{Ass_1}.\ L_{Ass_1} \ \vdash\ a \doteq c\ (Hyp)$
conclusions	$\ominus\ L_{Thm}.\ L_{Ass_1}, L_{Ass_2} \ \vdash\ even(a+b)\ (Open)$
binding	$\{L_3 \mapsto L_{Thm}, L_1 \to L_{Ass_1}, L_2 \to L_{Thm'}, f \mapsto even(a+b), \alpha \to \nu,$ $t \to a, t' \to c, pos \to < 1\ 1 >, f' \to even(c+b)\}$
constraints	$\emptyset$

Action	
method	=Subst-B
task	$L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$
premises	$\oplus\, L_{Thm'}.\ L_{Ass_1}, L_{Ass_2}\ \vdash\ even(a+c)\ (Open)$ $L_{Ass_2}.\ L_{Ass_2}\ \vdash\ b\dot{=}c\ (Hyp)$
conclusions	$\ominus\, L_{Thm}.\ L_{Ass_1}, L_{Ass_2}\ \vdash\ even(a+b)\ (Open)$
binding	$\{L_3 \mapsto L_{Thm}, L_1 \rightarrow L_{Ass_2}, L_2 \rightarrow L_{Thm'}, f \mapsto even(a+b), \alpha \rightarrow \nu,$ $t \rightarrow b, t' \rightarrow c, pos \rightarrow < 1\ 2 >, f' \rightarrow even(a+c)\}$
constraints	$\emptyset$

Finally, **ChooseAction** decides for one of the computed actions (step 7 in Figure 10). First, it rejects all actions that correspond to actions that have already been backtracked. This is done by the function *remove-backtracked*, which is applied to the current set of actions and the given history. If an action has the same given lines and the same binding as an action that is stored in the history as deleted action, then this action is removed from the alternative list. To the remaining actions **ChooseAction** applies the function *evalcrules-actions* to evaluate the control rules of kind 'Actions'. Provided the resulting list of actions is not empty, **ChooseAction** terminates and returns a pair consisting of the first element of the list of actions and the rest of the list of actions (i.e., the chosen action and the list of alternatives). If the list of actions is empty, then **ChooseAction** returns to the method selection point (step 2 in Figure 10) and repeats the sequence of matchings, application condition evaluation, outline computations evaluation, and outline completion for the next method of the method list. Similarly, **ChooseAction** returns to the method selection point and selects the next method, when the set of actions becomes empty during the matchings or by the evaluation of the application conditions. If **ChooseAction** fails to compute an action that does not correspond to a backtracked action and is not rejected by the control rules, then it terminates and returns **fail** (see step 2 in Figure 10).

3.5 Example

In the following, we shall explain the application of PLAN to a problem from the limit domain. Theorems of the *limit domain* make statements about the limit $\lim\limits_{x\to a} f(x)$ of a function f at a point a, about the limit $limseq\, X$ of a sequence X, about the continuity of a function f at a point a, and about the derivative of a function f at a point a. The standard definitions of limit, continuity, and derivative comprise so-called ϵ–δ criterions, i.e., proofs of such theorems postulate the existence of a δ such that a conjecture of the form $\ldots |X| < \epsilon$ is proved under assumptions of the form $\ldots |Y| < \delta$. For instance, the definition of $\lim$ in λ-notation is:

$$lim \ \equiv\ \lambda f.\lambda a.\lambda l.\forall \epsilon.(0 < \epsilon \Rightarrow \exists \delta.(0 < \delta \wedge \forall x.(|x-a| > 0 \wedge |x-a| < \delta \Rightarrow |f(x)-l| < \epsilon)))$$

An example theorem from the limit domain is LIM+ that states that the limit of the sum of two functions f and g equals the sum of their limits; that is, if $\lim\limits_{x\to a} f(x) = l_1$ and $\lim\limits_{x\to a} g(x) = l_2$ then $\lim\limits_{x\to a}(f(x) + g(x)) = l_1 + l_2$. When the definition of $\lim\limits_{x\to a}$ is expanded, the corresponding planning problem consists of two assumptions

$$\forall \epsilon_1. (0 < \epsilon_1 \Rightarrow \exists \delta_1. (0 < \delta_1 \wedge \forall x_1. (|x_1 - a| > 0 \wedge |x_1 - a| < \delta_1 \Rightarrow |f(x_1) - l_1| < \epsilon_1)))$$

and

$$\forall \epsilon_2. (0 < \epsilon_2 \Rightarrow \exists \delta_2. (0 < \delta_2 \wedge \forall x_2. (|x_2 - a| > 0 \wedge |x_2 - a| < \delta_2 \Rightarrow |g(x_2) - l_2| < \epsilon_2))).$$

And the theorem becomes

$$\forall \epsilon. (0 < \epsilon \Rightarrow \exists \delta. (0 < \delta \wedge \forall x. (|x - a| > 0 \wedge |x - a| < \delta \Rightarrow |(f(x) + g(x)) - (l_1 + l_2)| < \epsilon))).$$

Similar theorems in this class are LIM- and LIM* for the difference and the product of limits of functions. Moreover, there are corresponding theorems about continuity. Continuous+ states that the sum of two continuous functions is continuous, and Continuous- and Continuous* make similar statements for the difference and product of continuous functions.

When proving a limit theorem like LIM+, a δ has to be constructed that depends on an ϵ such that certain estimations hold. This is a non-trivial task for students as well as for traditional automated theorem provers.[16] The typical way a mathematician discovers a suitable δ is by incrementally restricting the possible values of δ. When proof planning limit theorems, PLAN adapts this approach by cooperating with the constraint solver $\mathcal{C}o\mathcal{SIE}$: (in)equality tasks that are simple enough for $\mathcal{C}o\mathcal{SIE}$ (i.e., tasks that are in the input language for $\mathcal{C}o\mathcal{SIE}$) are passed to $\mathcal{C}o\mathcal{SIE}$ and $\mathcal{C}o\mathcal{SIE}$ provides suitable instantiations for δ, when solutions for meta-variables are computed and inserted into the final proof plan.

For finding ϵ-δ-proofs, among others, the general methods $\exists$I-B, $\exists$E-F, $\forall$I-B, $\forall$E-F, $\wedge$I-B, $\wedge$E-F, $\Rightarrow$I-B, $\Rightarrow$E-F, SETFOCUS-B, and =Subst-B and the domain-specific methods TELLCS-B, TELLCS-F, ASKCS-B, COMPLEXESTIMATE-B, SOLVE*-B, SIMPLIFY-B, and SIMPLIFY-F are required. We introduced COMPLEXESTIMATE-B, ASKCS-B, TELLCS-B, TELLCS-F, $\forall$I-B, and $\exists$E-F already in section 2.6; =Subst-B is explained already in section 2.3. Similar to $\forall$I-B and $\exists$E-F also $\exists$I-B, $\forall$E-F, $\wedge$I-B, $\wedge$E-F, $\Rightarrow$I-B, and $\Rightarrow$E-F apply certain natural deduction rules. Actions of $\exists$I-B perform a backward $\exists_I$ step. They close a goal with formula $\exists x. P[x]$ and introduce a task whose goal has the formula $P[mv]$ in which x is replaced by a new meta-variable mv. Similarly, actions of $\forall$E-F perform a forward $\forall_E$ step and derive a new support $P[mv]$ with a new meta-variable mv from a given support $\forall x. P[x]$. Actions of $\wedge$I-B perform a backward $\wedge_I$ step and reduce a task whose goal has the formula $A_1 \wedge A_2$ to new tasks whose goals have the formulas A_1 and A_2. Actions of $\wedge$E-F perform the corresponding forward $\wedge_E$ decompositions on conjunctive support lines. Actions of $\Rightarrow$I-B perform a backward $\Rightarrow_I$ step and reduce a task with goal $A \Rightarrow B$ to a new task whose goal has the formula B and A as additional hypothesis. Moreover, A becomes the formula of a new support for this task. Actions of $\Rightarrow$E-F perform an $\Rightarrow_E$ step. When applied to a task with goal C and an support with formula $A \Rightarrow B$ they introduce two new tasks: a task with goal C, which contains also a new support with B as formula, and a task with goal A. Actions of the SOLVE*-B method exploit transitivity of $<, >, \leq, \geq$ and reduce a goal with formula $a_1 < b_1$ to a new task with formula $b_2\sigma \leq b_1\sigma$ in case a support $a_2 < b_2$ exists and a_1, a_2 can be unified by the substitution σ. Then, also a further new task is created whose formula is the conjunction of all mappings of the substitution σ (compare description of method COMPLEXESTIMATE-B in section 2.6). SIMPLIFY-B passes the goal of a given task to the com-

[16] BLEDSOE proposed in 1990 several versions of LIM+ as a challenge problem for automated theorem proving [7]. The simplest versions of LIM+ (problem 1 and 2 in [7]) are at the edge of the capabilities of traditional automated theorem provers but LIM* is certainly beyond their capabilities.

puter algebra system MAPLE and asks MAPLE to simplify it. If MAPLE succeeds, then the given task is reduced to a new task with the simplified formula. The analogous method SIMPLIFY-F derives a support with a simpler formula from a given support by calling MAPLE. Actions of SETFOCUS-B highlight a subformula in a support.

When applied to an ϵ–δ–problem, PLAN first decomposes the initial task with a complex formula into subtasks whose formulas are (in)equalities. This is done by actions that decompose formulas in tasks, e.g., actions of the methods $\wedge$I-B, $\forall$I-B, $\exists$I-B etc.

When faced with an inequality goal, PLAN first tries to apply the methods TELLCS-B and ASKCS-B, which both employ $\mathcal{C}o\mathcal{SIE}$. TELLCS-B passes the goal to $\mathcal{C}o\mathcal{SIE}$, whereas ASKCS-B asks $\mathcal{C}o\mathcal{SIE}$ whether the goal is entailed by its current constraints. If an inequality is too complex to be handled by $\mathcal{C}o\mathcal{SIE}$, then PLAN tries to apply methods that reduce an inequality to simpler inequalities. So, PLAN successively produces simpler inequalities, until it reaches inequalities that are accepted by $\mathcal{C}o\mathcal{SIE}$. This approach – handle with $\mathcal{C}o\mathcal{SIE}$ or simplify – is guided by the control rule `prove-inequality` given in Figure 3 in section 2.5, which is the central control rule to accomplish ϵ-δ-proofs with PLAN. In its IF-part `prove-inequality` checks whether the current goal is an inequality. If this is the case, it prefers the methods TELLCS-B, TELLCS-F, ASKCS-B, SIMPLIFY-B, SIMPLIFY-F, SOLVE*-B, COMPLEXESTIMATE-B, and SETFOCUS-B in this order.

In order to apply methods such as COMPLEXESTIMATE-B and SOLVE*-B unwrapping of (in)equality supports from the initial assumptions is necessary. This is realized as follows: First, PLAN applies SETFOCUS-B to highlight a promising subformula in a support (the application of SETFOCUS-B is suggested by `prove-inequality` if no other method is applicable, promising subformulas are chosen by another control rule guiding the supports and parameters choice point). Next, the highlighted subformula is unwrapped by actions that decompose supports, e.g., actions of the methods $\wedge$E-F, $\forall$E-F, $\exists$E-F etc.

Finally, when no task is left and PLAN invokes the function *employ-CS*, $\mathcal{C}o\mathcal{SIE}$ computes instantiations for the meta-variables that are consistent with the collected constraints.

Next, we briefly discuss the application of PLAN to the LIM+ problem.[17] PLAN first decomposes the initial theorem to tasks with the formulas $0 < mv_\delta$ and $|(f(c_x)+g(c_x))-(l_1+l_2)| < c_\epsilon$ where mv_δ is a meta-variable introduced for δ and c_x and c_ϵ are constants that replace x and ϵ, respectively. Moreover, the assumptions $0 < c_\epsilon$, $|c_x - a| > 0$, and $|c_x - a| < mv_\delta$ are created during the decomposition of the initial theorem and become supports of the new tasks. $0 < mv_\delta$ can be passed directly to $\mathcal{C}o\mathcal{SIE}$ by an action of TELLCS-B. $|(f(c_x) + g(c_x)) - (l_1 + l_2)| < c_\epsilon$ cannot be passed to $\mathcal{C}o\mathcal{SIE}$ directly. This triggers the decomposition of one of the two initial assumptions. If the initial assumption on f is decomposed, then PLAN obtains as new supports $0 < c_{\delta_1}$ and $|f(mv_{x_1}) - l_1| < mv_{\epsilon_1}$. Now PLAN can compute and introduce an action of COMPLEXESTIMATE-B using the latter new support line. During the evaluation of the application conditions of COMPLEXESTIMATE-B the substitution $mv_{x_1} \mapsto c_x$ is created and the computer algebra system MAPLE computes a decomposition $(f(c_x) + g(c_x)) - (l_1 + l_2) = 1*(f(c_x)-l_1)+(g(c_x)+l_2)$ (that is, the variables k and l of COMPLEXESTIMATE-B are bound to 1 and $g(c_x) - l_2$, respectively). Thus, the action of COMPLEXESTIMATE-B introduces new tasks with formulas $mv_{\epsilon_1} < \frac{c_\epsilon}{2*mv}$, $|1| \leq mv$, $0 < mv$, $|g(c_x) - l_2| < \frac{c_\epsilon}{2}$, and $mv_{x_1} \dot{=} c_x$. The formulas of the former three tasks and of the last one can all be passed directly to $\mathcal{C}o\mathcal{SIE}$ by actions of TELLCS-B. To deal with the remaining task with formula $|g(c_x) - l_2| < \frac{c_\epsilon}{2}$ PLAN decomposes

[17] A detailed description on how MULTI solves this problem is given in section 4.5.

the second initial assumption (on g) and derives new support lines with formulas $0 < c_{\delta_2}$ and $|g(mv_{x_2}) - l_2| < mv_{\epsilon_2}$. An action of SOLVE*-B reduces the goal with respect to the second new support to two new tasks with formulas $mv_{\epsilon_2} \leq \frac{c_\epsilon}{2}$ and $mv_{x_2} \doteq c_x$. Both tasks are closed by actions of TELLCS-B and their formulas are passed to $\mathcal{C}o\mathcal{SIE}$.

The decomposition of the initial assumptions results not only in the used support lines but also in tasks with the formulas $0 < mv_{\epsilon_1}$, $|mv_{x_1} - a| > 0$, $|mv_{x_1} - a| < c_{\delta_1}$ from the assumption on f and the analogue tasks from the assumption on g. The task $0 < mv_{\epsilon_1}$ is closed by the introduction of an action of TELLCS-B, which passes the formula to $\mathcal{C}o\mathcal{SIE}$. To close the other tasks PLAN introduces actions of the method SOLVE*-B that use the supports with formulas $|c_x - a| < mv_\delta$ and $|c_x - a| > 0$ (from the decomposition of the initial goal). The application of SOLVE*-B to the task $|mv_{x_1} - a| < c_{\delta_1}$ and the support $|c_x - a| < mv_\delta$ results in two new tasks with formulas $mv_\delta \leq c_{\delta_1}$ and $mv_{x_1} \doteq c_x$. The application of SOLVE*-B to the task $|mv_{x_1} - a| > 0$ and the support $|c_x - a| > 0$ results also in two new tasks with formulas $0 \leq 0$ and $mv_{x_1} \doteq c_x$. Whereas $0 \leq 0$ is closed by an actions of ASKCS-B the other three tasks are closed by actions of TELLCS-B, which pass their formulas to $\mathcal{C}o\mathcal{SIE}$. The corresponding tasks from the assumption on g are handled in the same way. Thereby the constraints $mv_\delta \leq c_{\delta_2}$, $mv_{x_2} \doteq c_x$, and $mv_{x_2} \doteq c_x$ are passed to $\mathcal{C}o\mathcal{SIE}$. Moreover, some actions of the TELLCS-F method during the planning process pass constraints in support lines to $\mathcal{C}o\mathcal{SIE}$: $0 < c_{\delta_1}$, $0 < c_{\delta_2}$, $0 < c_\epsilon$.

After propagating constraints, $\mathcal{C}o\mathcal{SIE}$ has the final constraint store in Figure 11. When asked for suitable instantiations for the meta-variables, $\mathcal{C}o\mathcal{SIE}$ provides the bindings $mv_{x_1} \mapsto c_x, mv_{x_2} \mapsto c_x, mv \mapsto 1$, $mv_{\epsilon_1} \mapsto \frac{c_\epsilon}{2}$, $mv_{\epsilon_2} \mapsto \frac{c_\epsilon}{2}$, and $mv_\delta \mapsto min(c_{\delta_1}, c_{\delta_2})$. These instantiations computed by $\mathcal{C}o\mathcal{SIE}$ are exactly the solutions that standard textbooks use for δ, ϵ_1, and ϵ_2 for LIM+.

$$
\begin{array}{lllll}
mv_{x_1} = c_x & & & & \\
mv_{x_2} = c_x & & & & \\
0 & < & c_{\delta_1} & < & +\infty \\
0 & < & c_{\delta_2} & < & +\infty \\
0 & < & c_\epsilon & < & +\infty \\
0 & < & mv_{\epsilon_1} & \leq & \frac{c_\epsilon}{2}, \frac{c_\epsilon}{2 * mv} \\
0 & < & mv_{\epsilon_2} & \leq & \frac{c_\epsilon}{2} \\
0 & < & mv_\delta & \leq & c_{\delta_1}, c_{\delta_2} \\
1 & \leq & mv & \leq & \frac{c_\epsilon}{2 * mv_{\epsilon_1}}
\end{array}
$$

Figure 11: The final constraint store of $\mathcal{C}o\mathcal{SIE}$ for LIM+.

PLAN can successfully plan all the challenge problems of BLEDSOE [7], i.e., the limit theorems LIM+, LIM-, LIM*, the theorems Continuous+, Continuous−, Continuous*, $\lim\limits_{x \rightarrow a} x = a$, $\lim\limits_{x \rightarrow a} c = c$, and the theorem that the composition of continuous functions is again continuous.

4 Proof Planning with Multiple Strategies

Proof planning with multiple strategies decomposes the previous monolithic proof planning process and replaces it by separated parameterized algorithms as well as different instances of these algorithms, so-called strategies. The strategies, which specify different behaviors of the algorithms, are the basic elements for proof construction in multiple-strategy proof planning. That is, the goal of multiple-strategy proof planning is to compute a sequence of strategy applications that derives a given theorem from a given set of assumptions. The decision on when to apply a strategy is not encoded once and forever into the system but rather is determined by meta-level reasoning using heuristic control knowledge of strategies and their combination.

In the following, we first introduce in section 4.1 the basic concepts of proof planning with multiple strategies and illustrate them with examples. Then, we describe in section 4.2 MULTI's blackboard architecture. Section 4.3 discusses the reasoning at the strategy-level with strategic control rules. We conclude with an informal description of all algorithms currently employed by MULTI that are not exemplified in section 4.1.

4.1 Algorithms, Strategies, and Tasks

Algorithms

MULTI enables the incorporation of heterogeneous, parameterized algorithms for different kinds of proof plan refinements and modifications. Currently, MULTI employs the following algorithms (technical descriptions of these algorithms, i.e., of the plan refinements or modifications they perform, are given in section 5):

PPLANNER refines a proof plan by introducing new actions.

INSTMETA refines a proof plan by instantiating meta-variables.

BACKTRACK modifies a proof plan by removing refinements of other algorithms.

EXP refines a proof plan by expanding complex steps.

ATP refines a proof plan by solving subproblems with traditional machine-oriented automated theorem provers.

CPLANNER refines a proof plan by transferring steps from a source proof plan or fragment.

The decomposition of PLAN allows to extend and generalize the functionalities of its subcomponents. This results in the independent and parameterized algorithms **PPLANNER**, **INSTMETA**, and **BACKTRACK** for action introduction, meta-variable instantiation, and backtracking. **EXP**, **ATP**, and **CPLANNER** integrate new refinements of the proof plan.

Strategies

Instances of these algorithms can be specified in different strategies. Technically, a *strategy* is a condition-action pair. The condition part states when the strategy is applicable. The action part consists of a modification or refinement algorithm and an instantiation of its parameters. Similar to the knowledge of the applicability of methods we separate the legal and heuristic knowledge of the applicability of strategies. The condition part of a strategy states the legal conditions that have to be satisfied in order for the strategy to be applicable, whereas *strategic control rules* reason about the heuristic utility of the application of strategies.

To execute or to apply a strategy means to apply its algorithm to the current proof planning state with respect to the parameter instantiation specified by the strategy. For instance, the parameters of **PPLANNER** are a set of methods, a list of control rules, a termination condition, and an action selection procedure. When MULTI executes a **PPLANNER** strategy, the **PPLANNER** algorithm introduces only actions that use the methods specified in the strategy. The actions are computed and selected by the action selection procedure (e.g., **CHOOSEACTION** or **CHOOSEACTIONALL**) specified by the strategy. The action selection procedures evaluate then the control rules specified by the strategy during the computation of actions. The application of the strategy terminates, when its termination condition is satisfied. Hence, different strategies of **PPLANNER** provide a means to structure the method and control rule knowledge. Both algorithms, **INSTMETA** and **BACKTRACK**, have one parameter. The parameter of **INSTMETA** is a function that determines how the instantiation for a meta-variable is computed. If MULTI applies a **INSTMETA** strategy with respect to a meta-variable mv, and if the computation function of the strategy yields a term t for mv, then **INSTMETA** substitutes mv by t in the proof plan. The parameter of **BACKTRACK** is a function that computes a set of refinement steps of other algorithms that have to be deleted. When MULTI applies a **BACKTRACK** strategy, then **BACKTRACK** removes all refinement steps that are computed by the function of the strategy as well as all steps that depend from these steps.

Notation Strategies are denoted in the sans serif font (e.g., NormalizeLineTask, UnwrapHyp).

Tasks

MULTI extends the task concept of PLAN. Since MULTI employs further kinds of tasks, the tasks used in PLAN (i.e., a pair consisting of an open line and its supports) are called *line-tasks* in MULTI. MULTI uses also *instantiation-tasks* and *expansion-tasks*. The introduction of a meta-variable into the plan results in an instantiation-task, that is, the task to instantiate this meta-variable. Similarly, the introduction of a method or tactic step into the $\mathcal{PDS}$, which is constructed during the proof planning process, results in an expansion-task, that is, the task to expand this step. An instantiation-task stores the meta-variable for which an instantiation has to be constructed. The instantiation task for meta-variable mv is written as $mv|^{Inst}$. An expansion-task consists of a proof line L in the $\mathcal{PDS}$, which is justified with a method or a tactic application. The expansion-task with line L is written as $L|^{Exp}$. MULTI stores all used kinds of tasks in an agenda.

Different tasks can be tackled by different algorithms and strategies. For instance, since strategies of **INSTMETA** introduce instantiations for meta-variables they can tackle instantiation-tasks. **EXP** is the suitable choice to deal with expansion-tasks, whereas strategies of **PPLANNER** or **ATP** can tackle line-tasks. A strategy checks in its condition part whether it is applicable to a particular task. That is, the condition of a strategy is a predicate on tasks. To *apply a strategy to a task* means to execute the strategy with respect to the task.

The algorithms and kinds of tasks currently employed by MULTI have been derived from the case studies. However, the MULTI framework is envisaged to be extended by further algorithms and further kinds of tasks, if needed.

Example Strategies

In the following, we describe some strategies needed to accomplish ϵ-δ-proofs (see section 3.5). The methods and control rules for ϵ-δ-proofs are structured into the three strategies NormalizeLineTask, UnwrapHyp, and SolveInequality. All three strategies are instantiations of **PPlanner**. A more detailed description of the application of these strategies and their cooperation when accomplishing ϵ-δ-proofs is given in section 4.5.

The strategy SolveInequality (see Table 2) is applicable to prove line-tasks whose formulas are inequalities or whose formulas can be reduced to inequalities. It comprises methods such as ComplexEstimate-B, TellCS-B, TellCS-F, AskCS-B, and Solve*-B (see section 3.5). Its list of control rules contains `prove-inequality` and `eager-instantiate`. Possible actions are computed and selected with the **ChooseAction** procedure. The strategy terminates, when there are no further line-tasks whose formulas are inequalities or whose formulas can be reduced to inequalities. Note that it is the parameterization of **PPlanner** that makes SolveInequality appropriate to tackle line-tasks whose formulas are inequalities as stated in the condition part of the strategy.

Strategy: SolveInequality		
Condition	*inequality-task*	
Action	Algorithm	**PPlanner**
	Action Procedure	**ChooseAction**
	Methods	ComplexEstimate-B, TellCS-B, TellCS-F, Solve*-B, AskCS-B ...
	C-Rules	`prove-inequality`, `eager-instantiate`, ...
	Termination	*no-inequalities*

Table 2: The SolveInequality strategy.

NormalizeLineTask (see Table 3) is used to decompose line-tasks whose goals are complex formulas with logical connectives and quantifiers. Typical methods in NormalizeLineTask are $\wedge$I-B and $\forall$I-B (see section 3.5). NormalizeLineTask employs the **ChooseAction** procedure for the action computation and selection and terminates, when all complex line-tasks are decomposed to literal line-tasks.

Strategy: NormalizeLineTask		
Condition	*complex-line-task*	
Action	Algorithm	**PPlanner**
	Action Procedure	**ChooseAction**
	Methods	$\forall$I-B, $\exists$I-B, $\wedge$I-B, ...
	C-Rules	
	Termination	*literal-line-tasks-only*

Table 3: The NormalizeLineTask strategy.

The aim of UnwrapHyp (see Table 4) is to unwrap a focused subformula of an assumption in order to make it available for proving a line-task. The list of its methods includes, for instance, $\forall$E-F and $\wedge$E-F. The control rule `tackle-focus` determines that, if UnwrapHyp is applied, then the actions of the available methods can be used only if they use a support in their premises that carries a focus and when their conclusions do not tackle the focused subformula. For instance, if a line-task has the supports $B_1 \wedge B_2$ and $A_1 \wedge (A_2 \wedge \mathit{focus}(A_3 \wedge A_4))$, then only actions of $\wedge$E-F that use the second support with the focus are allowed. The introduction of two actions of $\wedge$E-F derive the new support $\mathit{focus}(A_3 \wedge A_4)$ to which no further action of $\wedge$E-F can be applied since it would decompose the focused subformula. Similar to NormalizeLineTask and SolveInequality, UnwrapHyp uses the **ChooseAction** algorithm. It terminates as soon as all focused formulas are unwrapped.

Strategy: UnwrapHyp		
Condition	*focus-in-subformula*	
Action	Algorithm	**PPlanner**
	Action Procedure	**ChooseAction**
	Methods	$\forall$E-F, $\exists$E-F, $\wedge$E-F, ...
	C-Rules	`tackle-focus`
	Termination	*focus-at-top*

Table 4: The UnwrapHyp strategy.

In order to instantiate meta-variables that occur in constraints collected by $\mathcal{C}o\mathcal{SIE}$, we implemented the two **InstMeta** strategies InstIfDetermined and ComputeInstFromCS (see Table 5). InstIfDetermined is applicable only, if $\mathcal{C}o\mathcal{SIE}$ states that a meta-variable is already determined by the constraints collected so far. Then, the computation function connects to $\mathcal{C}o\mathcal{SIE}$ and receives this unique instantiation for the meta-variable. ComputeInstFromCS is applicable to all meta-variables for which constraints are stored in $\mathcal{C}o\mathcal{SIE}$. The computation function of this strategy requests from $\mathcal{C}o\mathcal{SIE}$ to compute an instantiation for a meta-variable that is consistent with all constraints collected so far.

Strategy: InstIfDetermined		
Condition	*determined-in-cs*	
Action	Algorithm	**InstMeta**
	Function	*get-determined-instantiation*

Strategy: ComputeInstFromCS		
Condition	*mv-in-cs*	
Action	Algorithm	**InstMeta**
	Function	*compute-consistent-instantiation*

Table 5: The **InstMeta** strategies InstIfDetermined and ComputeInstFromCS.

The dependency-directed backtracking described in section 3.3 is realized as the strategy BackTrackActionToTask (see Table 6) of the **BackTrack** algorithm. BackTrackActionToTask instantiates the **BackTrack** algorithm with the function *step-to-line-task*, which computes the action that introduced a line-task. BackTrackActionToTask is applicable to each line-task.

Strategy: BackTrackActionToTask		
Condition	*line-task*	
Action	Algorithm	**BACKTRACK**
	Function	*step-to-line-task*

Table 6: The BackTrackActionToTask strategy.

4.2 MULTI's Blackboard Architecture

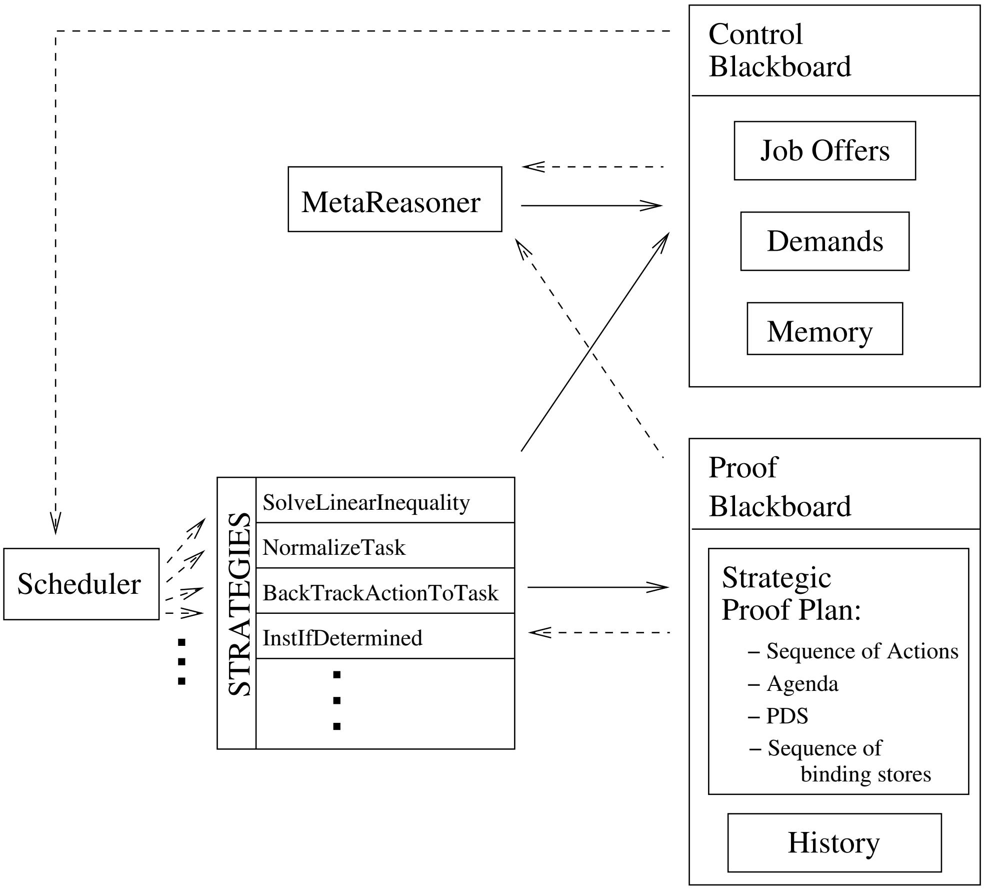


Figure 12: MULTI's blackboard architecture.

When we designed proof planning with multiple strategies, we aimed at a system that allows for the flexible cooperation of independent components for proof plan refinement and modification, guided by meta-reasoning. For the implementation we decided to use a blackboard architecture because this is an established means to organize the cooperation of independent components for solving a complex problem.

The fundamental ideas of the blackboard model [18] are (1) the segmentation of the knowledge base into modules that are kept separate and independent and (2) the separation of the inference engines that work on that knowledge. Each knowledge module can employ its own inference engine. The communication between the modules is limited to reading and writing in a common working memory, the *blackboard*. The blackboard can be further structured into regions that, for instance, contain different data structures. A basic blackboard architecture consists of a structured blackboard and the modular inference engine/knowledge base pairs which are called the *knowledge sources*.

The objective of each knowledge source is to contribute to the solution of the problem whose problem-solving state data are kept on the global blackboard. Control of knowledge source activation in blackboard systems is *data-directed* and *event-driven*. That is, the activation of the next knowledge source is determined by the changes of the data on the blackboard caused by other knowledge sources, rather than by explicit calls from other knowledge sources or some central sequencing mechanism. Knowledge sources check whether they are applicable with respect to the current solution state on the blackboard and indicate their applicability. Control modules choose the next knowledge source based on the solution state and on the existence of knowledge sources capable of improving the current state of the solution. As a result, blackboard systems do not rely on a pre-defined control of the application of the involved components but provide the flexibility to employ their knowledge sources opportunistically.

In the following, we give an informal overview on the MULTI system and the ideas behind it. A detailed technical description of the algorithms and concepts as well as a formal definition of strategic proof planning with MULTI are given in the next section.

MULTI's architecture is displayed in Figure 12. In this figure dashed arrows indicate information flow whereas solid arrows indicate that a knowledge source changes the content of the respective blackboard. MULTI's architecture is similar to the HEARSAY-III [19] and the BB1 [26] blackboard systems in that it employs two blackboards, the so-called *proof blackboard* and the *control blackboard*.

We decided for a two-blackboard architecture to emphasize the importance of both the solution of the proof planning problem whose status is stored on the proof blackboard and the solution of the control problem, that is, which possible strategy should the system perform next. The proof blackboard contains the current strategic proof plan, which consists of a sequence of actions, an agenda, a $\mathcal{PDS}$, and a sequence of binding stores, which store the collected instantiations of meta-variables, as well as the strategic history. The control blackboard contains three repositories to store information relevant for the control problem: job offers, demands, and a memory.

Corresponding to the two blackboards, there are also two sets of knowledge sources shown in Figure 12 that work on these blackboards. The strategies are the knowledge sources that work on the proof blackboard. A strategy can change the proof blackboard by refining or modifying the agenda, the $\mathcal{PDS}$, the history of strategies, and bindings of the meta-variables. The strategy component contains all the strategies that can be used. If a strategy's condition part is satisfied with respect to a certain task in the agenda, then the strategy posts its applicability with respect to this task as a job offer onto the control blackboard. Technically, a *job offer* is a pair (S, T) with a strategy S and a task T, which signs that T satisfies the condition of S. That is, in the terminology of blackboard systems, a task that satisfies the condition of a strategy is the event that triggers the strategy. The MetaReasoner is the knowledge source working on the control blackboard. It evaluates strategic control knowledge represented by strategic control rules in order to rank the job offers. The architecture contains a *scheduler* that checks the control blackboard, for its highest ranked job offer. Then, it executes the strategy of the job offer with respect to the task specified in the job offer. In a nutshell, MULTI operates according to the cycle in Figure 13, which passes the following steps:

Job Offer Strategies whose condition is true put a job offer onto the control blackboard.

Guidance The MetaReasoner evaluates the strategic control rules to order the job offers on the control blackboard.

Invocation A scheduler invokes the strategy who posed the highest ranked job offer.

Execution The algorithm of the invoked strategy is executed with respect to the parameter instantiation specified by the strategy.

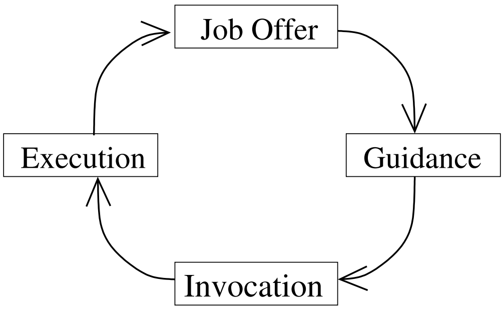


Figure 13: Cycle of MULTI.

Demands and the Memory

The choice of a job offer can depend on particular demand information issued by strategies onto the control blackboard and the content of the memory. An executed strategy can reason on whether it should interrupt. This can be sensible if the strategy is stuck or if it turns out that it should not proceed before another strategy is executed. Then, the execution of a strategy interrupts itself, places demands for other strategies onto the control blackboard, and stores a pair consisting of its execution status and the demands it posed in the memory. Interrupted executions of a strategy stored in the memory place job offers for their re-invocation onto the control blackboard. A job offer from the memory consists just of a pointer to the memory entry that posed this job offer. If such a job offer is scheduled, the interrupted strategy execution is re-invoked from the memory.

By posing demands and interrupting strategies particularly desired cooperations between strategies can be realized. For instance, in order to enable a flexible instantiation of meta-variables during the proof planning process (as opposed to PLAN's approach) the **INSTMETA** strategy InstIfDetermined and the **PPLANNER** strategy SolveInequality have to cooperate. This cooperation works as follows: The strategy SolveInequality contains the control rule `eager-instantiate`. If evaluated during an execution of SolveInequality, this control rule checks whether InstIfDetermined is applicable for an occurring meta-variable. If this is the case, it causes the interruption[18] of the execution of the SolveInequality strategy and poses the demand that InstIfDetermined should be applied with respect to the instantiation-task of the meta-variable. The status of the interrupted SolveInequality strategy is stored in the memory from where it can be reinvoked as soon as the posed demand is satisfied by the corresponding application of InstIfDetermined.

Binding Stores

[18] Interruption is an explicit choice point in the **PPLANNER** algorithm, see section 5.5.2.

MULTI allows to reason on existing meta-variables and possible instantiations for them. An equation of the form $mv_\alpha :=^b t_\alpha$ where mv_α is a meta-variable and t_α is a term of the same type α is called a *binding*. t is called the *instantiation* of the binding for mv. During the strategic proof planning process the current set of bindings is stored in a so-called *binding store*.

New bindings are not applied to existing proof lines in the constructed $\mathcal{PDS}$ or to proof lines in existing actions. Since the application of the bindings would replace occurrences of the meta-variables by occurrences of their current instantiations, it would not be possible to backtrack binding decisions in order to bind meta-variables differently (since the information on which subterms of the proof lines have been which meta-variables would have been lost). Rather, the current bindings are applied to copies of proof lines as soon as these are used. For instance, if a line-task has the task formula $|mv_x - c| < c_\delta$ and the current binding store contains the binding $mv_x :=^b c$, then **PPLANNER** applies the current binding to a copy of the task formula (see section 5.5.2 for details). The resulting formula, namely $|c - c| < c_\delta$, is then used in the action computation process instead of $|mv_x - c| < c_\delta$. Methods can become applicable wrt. the instantiated formula whereas they are not applicable wrt. the original formula with the meta-variables. For our example, a method for arithmetic simplifications becomes applicable and can reduce the formula $|c - c| < c_\delta$ to $0 < c_\delta$ which is not possible for $|mv_x - c| < c_\delta$. However, this step depends on the binding of mv_x; if this binding is removed (by backtracking the step that introduced the binding), then this step is not valid anymore.

MULTI constructs a sequence of binding stores in order to keep track of the dependencies between the changing bindings and the introduced actions. The introduction of a new binding creates a new binding store in the sequence. All following steps are performed with respect to this current binding store. When bindings are removed, then the binding store before the introduction of this binding is restored and all following binding stores are removed from the sequence. Moreover, all actions that potentially depend on the removed binding stores are deleted as well (for details see section 5.5.7 where backtracking in MULTI is described). We extended the notion of an action in proof planning for MULTI (see section 5.2). Actions have an additional slot **binding-store** in order to store a pointer to the binding store that was the current one when the action was computed.

Notation In the remainder of the report, the following symbols (maybe labeled with some subscripts or superscripts) are associated with the following objects:
$\mathcal{B}$S denotes a binding store,
$\vec{\mathcal{B}}$S denotes a sequence of binding stores.

4.3 Reasoning at the Strategy-Level

In the MULTI system, no order or combination of refinements or modifications on the proof blackboard is pre-defined. The choice of strategy applications results from meta-reasoning at the strategy-level that is conducted by the MetaReasoner, which evaluates the strategic control rules on the job offers on the control blackboard. Strategic control rules are formulated in the same control rule language as control rules on tasks, methods, supports and parameters, and actions (see section 2.5). They can reason about all information stored on the control blackboard and the proof blackboard (i.e., about the proof plan constructed so far and the plan process history) as well as about the mathematical domain of the proof planning problem.

The advantage of this knowledge-based control approach is that the control of MULTI can be easily extended and changed by modifying the strategic control rules. In contrast, when the combination of integrated components of a system is hard-coded into a control procedure, then each extension or change requires reimplementation of parts of the main control procedure. Moreover, the strategic control rules declaratively represent the heuristical control knowledge of how to combine the strategies of MULTI, so that this knowledge can be communicated to the user.

In the following, we shall discuss five strategic control rules, which are the backbone of the strategic control in MULTI.

```
(control-rule prefer-demand-satisfying-offers
              (kind strategic)
              (IF (job-offer-satisfies-demand JO))
              (THEN (prefer JO)))

(control-rule prefer-memory-offers
              (kind strategic)
              (IF (and (job-offer-from-memory JO)
                       (no-further-demands JO)))
              (THEN (prefer JO)))

(control-rule defer-memory-offers
              (kind strategic)
              (IF (and (job-offer-from-memory JO)
                       (further-demands JO)))
              (THEN (defer JO)))
```

Figure 14: The three strategic control rules `prefer-demand-satisfying-offers`, `prefer-memory-offers`, and `defer-memory-offers`.

The use of demands and the memory for the goal-directed cooperation of strategies is realized by the strategic control rules `prefer-demand-satisfying-offers`, `prefer-memory-offers`, and `defer-memory-offers` given in Figure 14. The rule `prefer-demand-satisfying-offers` states that, if a job offer on the control blackboard satisfies a demand on the control blackboard, then this job offer is preferred. Similarly, `prefer-memory-offers` states that, if there is a job offer from an interrupted strategy execution in the memory and all demands of this strategy execution are already satisfied, then this job offer should be preferred. `defer-memory-offers` defers job offers from interrupted strategy executions, if they have still unsatisfied demands.

The rules `prefer-backtrack-if-failure` and `reject-applied-offers` in Figure 15 realize a basic failure reasoning and the rejection of already applied strategies. The purpose of the `prefer-backtrack-if-failure` rule is to integrate backtracking with strategies of **PPLANNER**. When a **PPLANNER** strategy runs into a failure, that is, it encounters a line-task for which it finds no applicable action, then it interrupts and stores the status of its execution in the memory. `prefer-backtrack-if-failure` causes backtracking by preferring a job offer of the BackTrackActionToTask strategy with the line-task on which the execution of the **PPLANNER** strategy failed. Afterwards, the interrupted strategy execution can be re-invoked on

```
(control-rule reject-applied-offers
              (kind strategic)
              (IF (job-offer-already-applied JO))
              (THEN (reject JO)))

(control-rule prefer-backtrack-if-failure
              (kind strategic)
              (IF (and (algorithm-of-last-strategy-is PPLANNER)
                       (last-strategy-failure-on-line-task T)
                       (backtrack-job-offer-on JO T)))
              (THEN (prefer JO)))
```

Figure 15: The strategic control rules `reject-applied-offers` and `prefer-backtrack-if-failure`.

the changed proof blackboard. The idea behind `reject-applied-offers` is that a strategy that failed on a task should not be tried again on this task (although it is still applicable to the task, and, thus, it places a job offer onto the control blackboard). `reject-applied-offers` checks whether a job offer corresponds to a strategy execution that has already been tried but was backtracked later on. In this case, `reject-applied-offers` rejects the job offer.

The priority[19] of these control rules increases in the following order: `prefer-demand-satisfying-offers`, `prefer-memory-offers`, `defer-memory-offers`, `reject-applied-offers`, `prefer-backtrack-if-failure`. Although these control rules are the backbone of Multi's control, they realize only a default behavior and can be excluded by the user of Multi or can be overridden by other strategic control rules with higher priority.

4.4 Further Algorithms

The strategies **PPlanner**, **InstMeta**, and **BackTrack** are introduced and exemplified in section 4.1. Here we shall informally introduce the other three algorithms used in Multi, namely **Exp**, **ATP**, and **CPlanner**. Formal descriptions of all algorithms can be found in section 5.5.

Exp

The algorithm **Exp** tackles expansion-tasks. An expansion-task does not refer directly to an introduced action but contains a proof line in the constructed $\mathcal{PDS}$ whose justification is a complex step, that is, a method or a tactic application. For a proof line L with an abstract justification $(J\ P_1\ \ldots P_n)$ where J is a method or a tactic and $P_1,\ldots,P_n$ are the premises, **Exp** computes a proof segment, which derives L from $P_1,\ldots,P_n$ at a lower level of abstraction. If J is a method, then **Exp** computes the proof segment by instantiating the proof schema of J. If J is a tactic, then **Exp** evaluates the expansion function of J. Afterwards, **Exp** adds the new proof lines into the constructed $\mathcal{PDS}$ and adds a new justification to L at a lower level of abstraction.

Currently, the algorithm **Exp** is not parameterized. Since we distinguish technically between

[19] The MetaReasoner evaluates first the strategic control rules with lower priority. Since they are evaluated later on, the strategic control rules with higher priority cause the final changes of the alternative list of job offers.

a strategy and its algorithm we have implemented the strategy ExpS as the only strategy for the **Exp** algorithm. The application condition of ExpS states that this strategy is applicable to all expansion-tasks.

ATP

The algorithm **ATP** enables the application of automated theorem provers within MULTI in order to prove line-tasks. Its parameters are two functions for the application of an automated theorem prover (or several ones) and the check whether the obtained output is accepted as a proof. The first function obtains as input the line-task to which the **ATP** strategy is applied and returns the output of the employed ATP(s). The second function obtains the output of the ATPs and returns either *true* or *false* where true means that the function accepts the output as proof.

When a strategy of **ATP** succeeds for a line-task $L_{open} \blacktriangleleft \mathcal{SUPPS}_{L_{open}}$, then **ATP** closes the line L_{open} by the application of the tactic atp to the premises $\mathcal{SUPPS}_{L_{open}}$. Moreover, the output obtained from the application function of the strategy becomes the parameter of the justification. Whether this tactic application can be expanded depends on the accepted output. Currently, the expansion function of atp can deal with the following outputs:

- Resolution proofs from the provers OTTER [31], BLIKSEM [16], SPASS [63], PROTEIN [5], and equational proofs produced by the provers EQP [32] and WALDMEISTER [27]. On these outputs the expansion function of atp calls TRAMP [33], a proof transformation system that transforms resolution-style proofs into assertion level ND-proofs to be integrated into the $\mathcal{PDS}$.

- ND-proofs produced by TRAMP, if TRAMP is used as prover and not as transformation system (see below), and — with little transformational effort — ND-proofs provided by the higher-order prover TPS [3] (see [6] on what kind of transformations are necessary to incorporate TPS proofs into a $\mathcal{PDS}$).

Other output of automated theorem provers can be accepted by the respective strategies of **ATP** but cannot be further processed currently by the expansion function of the atp tactic.

Strategy: CallTramp		
Condition	*first-order-problem*	
Action	ATP Apply	*employ-tramp-on-task*
	ATP Output Check	*check-assertion-proof*

Table 7: The CallTramp strategy.

As example of a strategy of **ATP** consider CallTramp, which is depicted in Table 7. The application condition of CallTramp, *first-order-problem*, is satisfied by line-tasks, whose formulas are first-order. The application function, *employ-tramp-on-task*, employs TRAMP not as transformation module but as prover. This is possible since TRAMP cannot only transform the output of the connected provers but can also call these provers on a problem. When employed in this mode, TRAMP obtains a problem formalization, calls the connected automated theorem provers on the problem, and returns — if one of the connected provers succeeds — an assertion-level ND-proof.

The output check function of CallTramp, *check-assertion-proof*, checks whether the output provided by TRAMP is an ND-proof of the task.[20]

CPLANNER

Case-based reasoning is the approach to tackle new problems or subproblems by adapting given solutions or parts of given solutions of other problems or subproblems [10]. Case-based reasoning components for ΩMEGA were first developed as stand-alone systems not directly intertwined with the proof planner or other components. The last system developed in this paradigm was the TOPAL system [61, 46].

TOPAL obtains as input a *source proof plan* and a *target problem*. It successively transfers method applications from the source proof plan into a proof plan of the target problem. To do so, it computes and maintains possible mappings from objects of the source proof plan (e.g., tasks and proof lines) to corresponding objects of the target proof plan. With these mappings it computes new actions for the target proof plan from actions in the source proof plan. TOPAL processes the given source proof plan chronologically which means that TOPAL selects the actions to transfer in the order of the source proof plan.

The **CPLANNER** algorithm in MULTI extends TOPAL in several ways. First, **CPLANNER** is parameterized and enables the realization of different kinds of case-based reasoning. For instance, we realized a task-directed approach as an alternative to the chronological TOPAL approach. This task-directed approach, which is encoded in the **CPLANNER** strategy TaskDirectedAnalogy (see Table 8), first selects a task in the target proof plan and then selects an action to transfer in the source proof plan depending on the selected task. Second, **CPLANNER** allows not only for the transfer of method applications but also for the transfer of strategy applications from a strategic source proof plan into a strategic target proof plan. Moreover, the integration of **CPLANNER** into MULTI enables the flexible combination of case-based reasoning with the other algorithms in MULTI.

The parameters of **CPLANNER** are a list of so-called *action transfer procedures*, a list of control rules, and a termination condition. Action transfer procedures describe how source actions are transfered into target actions. The control rules guide the selection of action transfer procedures and interrupts. The termination condition specifies when the execution of the strategy terminates.

Technically, an action transfer procedure is a triple of a list of choice points, a list of instantiation functions, and a computation function. The choice points specify which objects have to be selected during the transfer process, the instantiation functions provide the alternative lists for the choice points, respectively, and the computation function computes either a new target action or a new demand for a tuple of selected objects. When the computation function provides a new target action, then **CPLANNER** introduces this action into the proof plan under construction. A demand causes **CPLANNER** to interrupt with this demand (see section 5.5.3 for details).

For instance, $TaskMeth$ is an action transfer procedure that realizes a task-directed transfer of source actions. $TaskMeth$ specifies the choice points target task, source action, target premises, and target parameters in this order. That is, it first selects the task in the target problem

[20] *check-assertion-proof* checks only whether the returned object is a proof tree whose root is the goal of the task and whose leaves are the supports of the task. It does not check whether each justification is correct since this would demand to expand the assertion-level proof.

to tackle and then selects the action to transfer in the source problem depending on this task. Finally, it chooses the target premises and target parameters depending on the selected target task and the selected source action. The computation function of $TaskMeth$ obtains the chosen objects as input and computes a new action of the method of the source action.

$TaskInst$ is an action transfer procedure for applications of **InstMeta** strategies. It first chooses an instantiation-task in the target plan. Next, it chooses an application of an **InstMeta** strategy in the source plan. Then, its computation function creates the demand to tackle the instantiation-task with the **InstMeta** strategy of the source action.

$TaskPPlanner$ is an action transfer procedure for applications of **PPlanner** strategies. It first chooses a line-task in the target proof plan and next an application of a **PPlanner** strategy in the source plan. The application of a **PPlanner** strategy essentially consists of a sequence of method actions (see section 5.2 for details). $TaskPPlanner$ reduces the transfer of the selected **PPlanner** strategy application to the transfer of the corresponding method action sequence. That is, it creates a demand for the recursive application of the **CPlanner** strategy TaskDirectedAnalogy with respect to the selected task and with the sequence of method actions as source actions.

The action transfer procedures $TaskMeth$, $TaskInst$, and $TaskPPlanner$ are combined in the **CPlanner** strategy TaskDirectedAnalogy, which is given in Table 8, in order to realize the task-directed transfer approach. The application condition of TaskDirectedAnalogy, *always-true-line+inst*, is satisfied by all line- and instantiation-tasks. The list of control rules is empty. The termination condition, *no-local-tasks*, is satisfied, when the initial task to which the strategy is applied and all tasks derived from this task are closed.

Strategy: TaskDirectedAnalogy		
Condition	*always-true-line+inst*	
Action	Action Trans. Procs.	$TaskMeth, TaskPPlanner, TaskInst$
	C-Rules	$\emptyset$
	Termination	*no-local-tasks*
	Source Actions	(free)

Table 8: The TaskDirectedAnalogy strategy

The applicability of TaskDirectedAnalogy is primarily not restricted by its condition part *always-true-line+inst*, but by its additional parameter, source actions, which is not a parameter of the algorithm **CPlanner**. Such additional parameters of strategies are called *free parameters*. They are not instantiated once and forever in the strategy. Rather, strategic control rules can suggest instantiations for a free parameter during the proof planning attempt.[21] A strategy with free parameters is applied only if a strategic control rule instantiates the free parameters.

The free parameter of TaskDirectedAnalogy, source actions, has to be instantiated by a strategic control rule with the sequence of source actions that the strategy should transfer.[22] A strategic control rule can choose, for instance, a complete source proof plan from a database of solved problems or it can choose a subsequence of actions of a given source proof plan. Instead of using actions of another problem (so-called *external analogy*) a strategic control rule can also

[21] Technically, strategies with free parameters post job offers, when their condition is satisfied and strategic control rules can then instantiate the free parameters by attaching instantiations to the job offer.

[22] The instantiation functions of the action transfer procedures look up the given source actions during the execution of the strategy and suggest then alternatives depending on these actions.

suggest a subsequence of actions of the proof plan under construction to be transfered to another part of the same proof plan (so-called *internal analogy*).

Examples and a detailed discussion of case-based reasoning in MULTI can be found in [55].

4.5 Example

To accomplish ϵ-δ-proof plans MULTI combines the **PPLANNER** strategies NormalizeLineTask, UnwrapHyp, and SolveInequality and the **INSTMETA** strategies InstIfDetermined and ComputeInstFromCS (see section 4.1), which interface $\mathcal{C}o\mathcal{SIE}$. In the following, we illustrate with the LIM+ example (introduced in section 3.5) how MULTI employs these strategies. However, before we elaborate the examples we discuss the employed strategies and their cooperation.

4.5.1 The Strategies and Their Cooperation

The strategy SolveInequality (see Table 2 in section 4.1) is central for accomplishing ϵ-δ-proofs with MULTI. It is applicable to prove line-tasks whose goals are inequalities or whose goals can be reduced to inequalities. A goal is reducible to inequalities if it contains defined terms whose unfolding will result in inequalities, for instance, lim, $limseq$, $cont$, and $deriv$. SolveInequality unfolds occurrences of these concepts both in the goal and in the supports of the task. The method for unfolding defined concepts in goals is DEFNUNFOLD-B, whereas DEFNUNFOLD-F unfolds defined concepts in supports. The central idea of SolveInequality to tackle inequality goals is similar to the approach of PLAN when accomplishing ϵ-δ-proofs (see section 3.5): pass to $\mathcal{C}o\mathcal{SIE}$ or simplify. Hence, also similar to PLAN's approach, the control rule `prove-inequality` given in Figure 3 in section 2.5 is central in SolveInequality.

SolveInequality comprises the knowledge of how to deal with inequalities and with problems that can be reduced to inequalities. As opposed thereto, the strategies NormalizeLineTask and UnwrapHyp comprise the domain-independent, general knowledge of how to decompose complex formulas with logical connectives and quantifiers. SolveInequality decides once for the decomposition of a complex goal or the unwrapping of a subformula from a complex support. Then, it switches to NormalizeLineTask or UnwrapHyp, which perform all single decomposition steps. This saves SolveInequality from reasoning permanently on the application of methods that decompose single logical connectives and quantifiers such as $\wedge$I-B or $\wedge$E-F.

Technically, the cooperation between SolveInequality and NormalizeLineTask and UnwrapHyp works as follows. For line-tasks whose goals are complex formulas that contain inequality subformulas (e.g., goals that arise from unfolding lim, $limseq$, $cont$, or $deriv$) SolveInequality interrupts and places a demand for the strategy NormalizeLineTask on the control blackboard. Guided by this demand, MULTI invokes NormalizeLineTask, which decomposes the complex goal. When re-invoked by MULTI, SolveInequality can tackle the inequalities in the resulting goals. The switch from SolveInequality to UnwrapHyp is driven by missing support inequalities, which are needed for the application of the methods COMPLEXESTIMATE-B and SOLVE*-B. If the other methods preferred by `prove-inequality` fail, then the application of SETFOCUS-B highlights a subformula in an existing support. Afterwards, SolveInequality interrupts and places a demand for the invocation of UnwrapHyp to unwrap the highlighted subformula. When the subformula is unwrapped, SolveInequality can continue with a new support that may enable further steps. The application of SETFOCUS-B (i.e., the selection of the support and the subformula to highlight) is guided by the control rule `choose-unwrap-support` for the supports and parameters choice point. `choose-unwrap-support` analyzes the supports of the task on which the other methods are not applicable. It searches for inequality subformulas in the supports that are similar to the goal of the task. The idea is that similar formulas are likely to unify with the goal such that COMPLEXESTIMATE-B and SOLVE*-B become applicable.

To accomplish ϵ-δ-proofs plans also two **INSTMETA** strategies, namely ComputeInstFromCS and InstIfDetermined, are used that interface the constraint solver $\mathcal{CoSIE}$. Whereas InstIfDetermined asks $\mathcal{CoSIE}$ for instantiations of meta-variables that are already determined by the collected constraints, ComputeInstFromCS asks $\mathcal{CoSIE}$ to compute instantiations for the occurring meta-variables that are consistent with the collected constraints.

The invocation of ComputeInstFromCS is delayed by the strategic control rule `delay-ComputeInstCosie` until all line-tasks are closed. This delay of the computation of instantiations for meta-variables is sensible, since the instantiations should not be computed before all constraints are collected, that is, not before all line-tasks are closed. However, when the current constraints already determine a meta-variable, then a further delay of the corresponding instantiation is not necessary. Rather, immediate instantiations of determined meta-variables can simplify a problem (e.g., see [35]).

To enable the flexible instantiation of determined meta-variables SolveInequality cooperates with the strategy InstIfDetermined. Technically, this works as follows. When $\mathcal{CoSIE}$ signals that a meta-variable is determined, then the control rule `eager-instantiate` in SolveInequality fires. It interrupts SolveInequality and places a demand for InstIfDetermined with respect to the determined meta-variable. After the introduction of a binding for the meta-variable by InstIfDetermined MULTI re-invokes SolveInequality.

4.5.2 The LIM+ Example with MULTI

The LIM+ problem states that the limit of the sum of two functions f and g equals the sum of their limits. That is, the problem states that

$$\begin{array}{rl} & LIM+\text{: } \lim_{x\to a}(f(x)+g(x)) = l_f + l_g \\ \text{follows from} & Lim_f\text{: } \lim_{x\to a} f(x) = l_f \\ \text{and} & Lim_g\text{: } \lim_{x\to a} g(x) = l_g. \end{array}$$

Figure 16 and Figure 17 show the interesting parts, i.e., the parts created by SolveInequality, of the resulting $\mathcal{PDS}$. We indicate the contributions of NormalizeLineTask and UnwrapHyp by justifications in the $\mathcal{PDS}$ such as (UnwrapHyp L_3) in line L_{49}, which abbreviate the proof segments created by these strategies. The complete $\mathcal{PDS}$ is given in appendix B. Note that we describe the proof planning process in progress. Hence, we introduce meta-variables, when they arise. When there is a binding for a meta-variable during the proof planning process, then the proof lines created after the introduction of the binding use the instantiation of the meta-variable in order to clarify the following computations.

The proof planning process starts with the invocation of SolveInequality on the initial task $LIM+ \blacktriangleleft \{Lim_f, Lim_g\}$. SolveInequality first unfolds the occurrences of lim. Afterwards, it switches to NormalizeLineTask, which decomposes the resulting complex goal in line L_1 into the goals $|(f(c_x)+g(c_x))-(l_f+l_g)| < c_\epsilon$ in L_{12} and $0 < mv_\delta$ in L_8 where c_ϵ and c_x are

Lim_f.	Lim_f	$\vdash \lim\limits_{x \to a} f(x) = l_f$	(Hyp)
Lim_g.	Lim_g	$\vdash \lim\limits_{x \to a} g(x) = l_g$	(Hyp)
L_2.	Lim_f	$\vdash \forall \epsilon_1 . (0 < \epsilon_1 \Rightarrow \exists \delta_1 . (0 < \delta_1 \wedge \forall x_1 . (\|x_1 - a\| < \delta_1 \wedge \|x_1 - a\| > 0 \Rightarrow \|f(x_1) - l_f\| < \epsilon_1)))$	(DefnUnfold-F Lim_f)
L_3.	Lim_g	$\vdash \forall \epsilon_2 . (0 < \epsilon_2 \Rightarrow \exists \delta_2 . (0 < \delta_2 \wedge \forall x_2 . (\|x_2 - a\| < \delta_2 \wedge \|x_2 - a\| > 0 \Rightarrow \|g(x_2) - l_g\| < \epsilon_2)))$	(DefnUnfold-F Lim_g)
L_{21}.	L_{21}	$\vdash 0 < c_{\delta_1} \wedge \forall x_1 . (\|x_1 - a\| < c_{\delta_1} \wedge \|x_1 - a\| > 0 \Rightarrow \|f(x_1) - l_f\| < mv_{\epsilon_1})$	(Hyp)
L_{42}.	L_{42}	$\vdash 0 < c_{\delta_2} \wedge \forall x_2 . (\|x_2 - a\| < c_{\delta_2} \wedge \|x_2 - a\| > 0 \Rightarrow \|g(x_2) - l_g\| < mv_{\epsilon_2})$	(Hyp)
L_{11}.	L_{11}	$\vdash \|c_x - a\| > 0 \wedge \|c_x - a\| < mv_\delta$	(Hyp)
L_5.	L_5	$\vdash 0 < c_\epsilon$	(Hyp)
L_{52}.	$\mathcal{H}_2$	$\vdash mv_{x_2} \doteq c_x$	(TellCS-B)
L_{53}.	$\mathcal{H}_2$	$\vdash mv_{\epsilon_2} \leq \frac{1}{2} * c_\epsilon$	(TellCS-B)
L_{49}.	$\mathcal{H}_2$	$\vdash \|g(mv_{x_2}) - l_g\| < mv_{\epsilon_2}$	(UnwrapHyp L_3)
L_{48}.	$\mathcal{H}_2$	$\vdash \|g(c_x) - l_g\| < \frac{1}{2} * c_\epsilon$	(Solve*-B L_{49} L_{52} L_{53})
L_{37}.	$\mathcal{H}_1$	$\vdash \|g(c_x) - l_g\| < \frac{1}{2} * c_\epsilon$	(UnwrapHyp L_3 L_{48} L_{39} L_{50} L_{51})
L_{31}.	$\mathcal{H}_1$	$\vdash \|1\| \leq mv$	(TellCS-B)
L_{32}.	$\mathcal{H}_1$	$\vdash mv_{\epsilon_1} \leq \frac{c_\epsilon}{2 * mv}$	(TellCS-B)
L_{33}.	$\mathcal{H}_1$	$\vdash \|g(c_x) - l_g\| < \frac{c_\epsilon}{2}$	(Simplify-B L_{37})
L_{34}.	$\mathcal{H}_1$	$\vdash 0 < mv$	(TellCS-B)
L_{35}.	$\mathcal{H}_1$	$\vdash mv_{x_1} \doteq c_x$	(TellCS-B)
L_{28}.	$\mathcal{H}_1$	$\vdash \|f(mv_{x_1}) - l_f\| < mv_{\epsilon_1}$	(UnwrapHyp L_2)
L_{27}.	$\mathcal{H}_1$	$\vdash \|((f(c_x) + g(c_x)) - l_f) - l_g\| < c_\epsilon$	(ComplexEstimate-B L_{28} L_{31} L_{32} L_{33} L_{34} L_{35})
L_{16}.	$\mathcal{H}_3$	$\vdash \|((f(c_x) + g(c_x)) - l_f) - l_g\| < c_\epsilon$	(UnwrapHyp L_2 L_{27} L_{18} L_{29} L_{30})
L_{12}.	$\mathcal{H}_3$	$\vdash \|(f(c_x) + g(c_x)) - (l_f + l_g)\| < c_\epsilon$	(Simplify-B L_{16})
L_8.	$\mathcal{H}_4$	$\vdash 0 < mv_\delta$	(TellCS-B)
L_1.	Lim_f, Lim_g	$\vdash \forall \epsilon . (0 < \epsilon \Rightarrow \exists \delta . (0 < \delta \wedge \forall x . (\|x - a\| < \delta \wedge \|x - a\| > 0 \Rightarrow \|(f(x) + g(x)) - (l_f + l_g)\| < \epsilon)))$	(NormalizeLineTask L_8 L_{12})
$LIM+$.	Lim_f, Lim_g	$\vdash \lim\limits_{x \to a} (f(x) + g(x)) = l_f + l_g$	(DefnUnfold-B L_1)

$\mathcal{H}_1 = \{Lim_f, Lim_g, L_5, L_{11}, L_{21}\}$, $\mathcal{H}_2 = \{Lim_f, Lim_g, L_5, L_{11}, L_{21}, L_{42}\}$
$\mathcal{H}_3 = \{Lim_f, Lim_g, L_5, L_{11}\}$, $\mathcal{H}_4 = \{Lim_f, Lim_g, L_5\}$

Figure 16: ϵ-δ-proof for LIM+ (part I).

constants introduced for the universally quantified variables ϵ and x in L_1 and mv_δ is a meta-variable introduced for the existentially quantified variable δ.

Both new goals are inequalities and SolveInequality tackles them guided by the control rule `prove-inequality`. It closes $0 < mv_\delta$ directly by an application of TellCS-B, which passes the formula to $\mathcal{CoSIE}$. $|(f(c_x) + g(c_x)) - (l_f + l_g)| < c_\epsilon$ is not accepted by $\mathcal{CoSIE}$ and therefore TellCS-B is not applicable. SolveInequality simplifies this goal to $|((f(c_x) + g(c_x)) - l_f) - l_g| < c_\epsilon$ in line L_{16} but then fails to solve this goal with the given supports.

choose-unwrap-support detects the subformula $|f(x_1) - l_f| < \epsilon_1$ of L_2 as a promising support and guides the application of the method SetFocus-B to highlight the subformula. This triggers the interruption of SolveInequality and the invocation of UnwrapHyp for this subformula. The application of UnwrapHyp yields the new support $|f(mv_{x_1}) - l_f| < mv_{\epsilon_1}$ in line L_{28}, but also the three new goals $0 < mv_{\epsilon_1}$ in line L_{18}, $|mv_{x_1} - a| < c_{\delta_1}$ in L_{29}, and $|mv_{x_1} - a| > 0$ in L_{30}. Here UnwrapHyp introduces the constant c_{δ_1} for the existentially quantified variable δ_1 and the meta-variables mv_{ϵ_1} and mv_{x_1} for the universally quantified variables ϵ_1 and x_1 in L_2.

When SolveInequality is re-invoked, it can apply ComplexEstimate-B to the goal $|((f(c_x) + g(c_x)) - l_f) - l_g| < c_\epsilon$ and the new support $|f(mv_{x_1}) - l_f| < mv_{\epsilon_1}$. This results in the five new goals $|1| \leq mv$ in L_{31}, $mv_{\epsilon_1} \leq \frac{c_\epsilon}{2*mv}$ in L_{32}, $|g(c_x) - l_g| < \frac{c_\epsilon}{2}$ in L_{33}, $0 < mv$ in L_{34}, and $mv_{x_1} \doteq c_x$ in L_{35}. Except L_{33} all goals are closed by applications of TellCS-B, which pass the respective formulas as constraints to $\mathcal{CoSIE}$. Since $mv_{x_1} \doteq c_x$ determines mv_{x_1} in $\mathcal{CoSIE}$ the control rule eager-instantiate fires and interrupts SolveInequality. Its demand causes Multi to invoke InstIfDetermined on the instantiation-task of mv_{x_1}. InstIfDetermined introduces the binding $mv_{x_1} :=^b c_x$ into the strategic proof plan.

The re-invoked SolveInequality simplifies $|g(c_x) - l_g| < \frac{c_\epsilon}{2}$ to $|g(c_x) - l_g| < \frac{1}{2} * c_\epsilon$ in L_{37} but then fails on this goal with the existing supports. choose-unwrap-support detects the subformula $|g(x_2) - l_g| < \epsilon_2$ of L_3 as a promising support and guides the corresponding application of the method SetFocus-B to highlight this subformula. Afterwards, SolveInequality interrupts and Multi switches to UnwrapHyp, which unwraps the subformula and yields the new support $|g(mv_{x_2}) - l_g| < mv_{\epsilon_2}$ in line L_{49}. The unwrapping yields also the three new goals $0 < mv_{\epsilon_2}$ in line L_{39}, $|mv_{x_2} - a| < c_{\delta_2}$ in L_{50}, and $|mv_{x_2} - a| > 0$ in L_{51}. UnwrapHyp introduces the constant c_{δ_2} for the existentially quantified variable δ_2 and the meta-variables mv_{ϵ_2} and mv_{x_2} for the universally quantified variables ϵ_2 and x_2 in L_3.

When re-invoked, SolveInequality applies Solve*-B to the goal $|g(c_x) - l_g| < \frac{1}{2} * c_\epsilon$ and the new support $|g(mv_{x_2}) - l_g| < mv_{\epsilon_2}$. This results in the new goals $mv_{x_2} \doteq c_x$ in L_{52} and $mv_{\epsilon_2} \leq \frac{1}{2} * c_\epsilon$ in L_{53}, which SolveInequality closes by TellCS-B. $mv_{x_2} \doteq c_x$ determines the meta-variable mv_{x_2} in $\mathcal{CoSIE}$. Thus, the control rule eager-instantiate suggests a switch from SolveInequality to InstIfDetermined, which introduces the binding $mv_{x_2} :=^b c_x$ into the strategic proof plan.

$L_{18}.$	$\mathcal{H}_3$	$\vdash 0 < mv_{\epsilon_1}$	(TellCS-B)
$L_{39}.$	$\mathcal{H}_3$	$\vdash 0 < mv_{\epsilon_2}$	(TellCS-B)
$L_{11}.$	L_{11}	$\vdash \lvert c_x - a\rvert > 0 \wedge \lvert c_x - a\rvert < mv_\delta$	(Hyp)
$L_{14}.$	L_{11}	$\vdash \lvert c_x - a\rvert > 0$	($\wedge$E-F L_{11})
$L_{13}.$	L_{11}	$\vdash \lvert c_x - a\rvert < mv_\delta$	($\wedge$E-F L_{11})
$L_{61}.$	$\mathcal{H}_1$	$\vdash 0 \leq 0$	(AskCS-B)
$L_{59}.$	$\mathcal{H}_1$	$\vdash mv_\delta \leq c_{\delta_1}$	(TellCS-B)
$L_{57}.$	$\mathcal{H}_2$	$\vdash 0 \leq 0$	(AskCS-B)
$L_{55}.$	$\mathcal{H}_2$	$\vdash mv_\delta \leq c_{\delta_2}$	(TellCS-B)
$L_{29}.$	$\mathcal{H}_1$	$\vdash \lvert mv_{x_1} - a\rvert < c_{\delta_1}$	(Solve*-B L_{13} L_{59})
$L_{30}.$	$\mathcal{H}_1$	$\vdash \lvert mv_{x_1} - a\rvert > 0$	(Solve*-B L_{14} L_{61})
$L_{50}.$	$\mathcal{H}_2$	$\vdash \lvert mv_{x_2} - a\rvert < c_{\delta_2}$	(Solve*-B L_{13} L_{55})
$L_{51}.$	$\mathcal{H}_2$	$\vdash \lvert mv_{x_2} - a\rvert > 0$	(Solve*-B L_{14} L_{57})

$\mathcal{H}_1 = \{Lim_f, Lim_g, L_5, L_{11}, L_{21}\}$, $\mathcal{H}_2 = \{Lim_f, Lim_g, L_5, L_{11}, L_{21}, L_{42}\}$

$\mathcal{H}_3 = \{Lim_f, Lim_g, L_5, L_{11}\}$, $\mathcal{H}_4 = \{Lim_f, Lim_g, L_5\}$

Figure 17: ϵ-δ-proof for LIM+ (part II).

Afterwards, SolveInequality has to deal with the remaining goals L_{18}, L_{29}, L_{30}, and L_{39}, L_{50}, L_{51}, which resulted from the applications of the UnwrapHyp strategy. Figure 17 gives the $\mathcal{PDS}$ segment created by SolveInequality for these goals. It closes L_{18} and L_{39} directly by TELLCS-B. The inequalities in the other goals cannot be passed to $\mathcal{CoSIE}$ directly because TELLCS-B is not applicable to them. Instead, SolveInequality applies SOLVE*-B to these goals with supports that stem from the decomposition of the initial goal by NormalizeLineTask. The applications of SOLVE*-B result in inequality goals, which SolveInequality closes either with TELLCS-B or ASKCS-B.

After closing all line-tasks, SolveInequality terminates. Next, MULTI invokes ComputeInstFromCS on the instantiation-tasks and $\mathcal{CoSIE}$ provides instantiations for the meta-variables that are consistent with the collected constraints (see Figure 11 in section 3.5). ComputeInstFromCS inserts these instantiations as the bindings

$$mv{:=}^b 1,\, mv_{\epsilon_1}{:=}^b \tfrac{c_\epsilon}{2},\, mv_{\epsilon_2}{:=}^b \tfrac{c_\epsilon}{2},\text{ and } mv_\delta{:=}^b min(c_{\delta_1}, c_{\delta_2})$$

into the strategic proof plan.

5 Formal Description of MULTI

In the previous section, we explained the design of MULTI and its basic concepts. In this section, we shall give a formal description of MULTI.

Proof planning with multiple strategies computes strategic actions and introduces them into a strategic proof plan. A strategic action is the instantiation of a strategy pattern corresponding to method actions, which are instantiations of methods. Similar to proof plans in PLAN a strategic proof plan consists of a sequence of actions, an agenda, and a $\mathcal{PDS}$. Strategic proof plans contain additionally a sequence of so-called binding stores to keep track of introduced meta-variable instantiations.

The structure of the section is as follows. First, we introduce some new data structures used by MULTI. In section 5.2, we describe the different kinds of strategic actions in MULTI. Afterwards, we formally describe strategic proof plans and give the operational semantics of strategic actions in section 5.3. Section 5.4 describes the strategic manipulation records, which MULTI uses to construct a history. After the introduction of all necessary elements, we describe MULTI's main cycle and the modification and refinement algorithms integrated so far in section 5.5. We conclude this section with the discussion of some particular technical features of MULTI in section 5.6.

5.1 New Data Structures

In this section, we discuss some new data structures used in MULTI and their role during the strategic proof planning process.

Task Tags

In MULTI, a strategy is executed with respect to a particular task (from the blackboard point of view we can say that the existence of the task triggers the invocation of the strategy). A particular execution of a strategy tackles then the task by which it was triggered rather than arbitrary tasks. This is easy to realize for the algorithms **EXP**, **ATP**, and **INSTMETA** since these algorithms perform just one refinement step before they terminate. The situation is more complicated for the algorithms **PPLANNER** and **CPLANNER** since they may perform a sequence of proof plan modifications (e.g., introduce several actions) before they terminate or interrupt. When applied with respect to an initial task, these algorithm should tackle this task and tasks that are derived from it but they should ignore other tasks in the agenda. Moreover, if a strategy execution of **CPLANNER** or **PPLANNER** interrupts and other strategies are executed, then some of these strategies work on tasks created by the interrupted strategy some of them work on other tasks. When the initial strategy is re-invoked again, then it should tackle tasks derived from its own tasks but it should ignore other tasks created meanwhile. To organize this behavior a maintenance mechanism is needed, which keeps track of which tasks are relevant for which strategies.

In MULTI, the desired behavior is supported by so-called *task tags*. When a strategy of **CPLANNER** and **PPLANNER** is invoked, then it creates a new task tag $@_T$, which uniquely refers to this execution of the strategy. The task tag is pinned to the task that triggered the strategy. When a proof plan modification in MULTI reduces a task to some new tasks, then the new tasks inherit all tags from the initial one. An execution of a strategy of **CPLANNER** or **PPLANNER** considers only tasks that carry its tag. When the strategy execution terminates, then its tag is removed from all tasks. When a strategy execution interrupts and is re-invoked later on, then the re-invocation continues to work with the task tag created by the initial invocation.

If used in several not-terminated strategies, then one task can carry several tags. For instance, when an execution of a **PPLANNER** strategy creates a task T, then T carries the tag of this execution. Afterwards, the execution interrupts and a different strategy is applied to T. Then, this second strategy execution creates a new tag, which is also pinned to T. All actions introduced by this second strategy execution inherit both tags of T. When the second strategy execution terminates and its tag is removed, then the resulting tasks carry still the tag of the first strategy execution. Thus, when the first strategy execution is re-invoked, it can continue to tackle these tasks.

Note that the task tags describe only which tasks can be tackled by a strategy execution. This does not mean that the other tasks are “invisible” or temporarily removed. Control rules evaluated by **CPLANNER** and **PPLANNER** can reason on all tasks of the current agenda.

Execution Messages

When a strategy execution stops, then its result and the reason why it stops are relevant information for MULTI since MULTI treats different kinds of termination differently (see section 5.5). Moreover, this information is important for the meta-reasoning with strategic control rules. Therefore, each strategy execution in MULTI stops with a so-called *execution message*, which contains the available termination information. So far, MULTI uses the following execution messages:

- A *success message* occurs when the strategy execution is successful on the given task.

- A *failure message* occurs when the strategy execution fails on the given task because of some problems (e.g., a strategy of **PPLANNER** fails because there are no further applicable actions).

- An *interruption message* occurs when a strategy of **CPLANNER** or **PPLANNER** is interrupted.

The algorithms can attach further information to the execution messages, which can also be used by the strategic control rules. For instance, an algorithm can attach information on what kind of failure occurred to a failure message (see section 5.6.5).

Execution messages are stored in the history entries created by the strategy executions (see section 5.4). When which algorithm terminates with which execution message is described in detail in section 5.5. When a strategy execution terminates with a success message we also say that *the application of the strategy was successful.*

Demands and Memory Entries

For the algorithms **CPLANNER** and **PPLANNER** a strategy execution can interrupt. If this is the case, the strategy execution creates so-called demands and adds them to the demand repository on the control blackboard. MULTI knows for the following *demands*:

- A demand $S - ON - T$, which specifies a strategy S and a task T, is called a *strategy-task-demand*. This demand is satisfied by a successful application of the strategy S to the task T.
- A demand $S - ON - ?$, which specifies a strategy S but no task, is called a *strategy-demand*. This demand is satisfied by a successful application of the strategy S to any task.
- A demand $? - ON - T$, which specifies a task T but no strategy, is called a *task-demand*. This demand is satisfied by a successful application of any strategy to the task T.

An interrupted strategy execution writes also an entry into the memory repository on the control blackboard. A *memory entry* is a pair $(@_T, \{P_{D_1}, \ldots, P_{D_n}\})$ of a task tag $@_T$ and a set of pointers $\{P_{D_1}, \ldots, P_{D_n}\}$ to the demands of the interrupted strategy execution in the demands repository. MULTI uses the $@_T$ to re-invoke the strategy execution later on (see section 5.5.2 for details). Moreover, it makes use of the pointers to check whether the demands of the interrupted strategy are satisfied such that the strategy execution can be re-invoked again (see section 5.5.1 for details).

5.2 Strategic Actions

PLAN computes and introduces actions into a proof plan. An action is an instantiation of a method, which is a pattern of a proof step (see section 2.4). To extend this approach of action computation and introduction to strategic proof planning there is a strategic pattern associated with each algorithm in MULTI (except **BACKTRACK**). The application of a strategy computes an instantiation of the pattern of its algorithm, a so-called *strategic action*, and introduces it into the strategic proof plan.

In this section, we shall describe the strategic actions created by **PPLANNER**, **INSTMETA**, **EXP**, **ATP**, and **CPLANNER**. The algorithm **BACKTRACK** does not create actions but deletes actions of other algorithms. Note that, henceforth, we call instantiations of methods *method actions* in order to distinguish them from the different strategic actions, which we call **PPlanner** *actions*, **InstMeta** *actions*, **Exp** *actions*, **ATP** *actions*, and **CPlanner** *actions*.

Technically, strategic actions are implemented as frame data structures. Each strategic action has the slots **strategy**, **task**, and **binding-store**. The *strategy of an action* and the *task of an action* are pointers to the strategy and the task with respect to which the action was computed. The *binding store of an action* is a pointer to the binding store, which was the current binding store, when the action was computed. Depending on the algorithm the different strategic actions have also further slots.

PPlanner and CPlanner

The algorithms **PPlanner** and **CPlanner** successively introduce actions into a strategic proof plan, **PPlanner** with respect to a given set of methods and control rules, **CPlanner** with respect to a given plan or a given plan fragment. Thus, actions of **PPlanner** and **CPlanner** are essentially abstractions of the sequence of actions introduced by the respective algorithm. The sequence of introduced actions is stored in the slot **action-sequence** of a **PPlanner** or **CPlanner** action.

Executions of **PPlanner** and **CPlanner** strategies can interrupt and can be re-invoked later on. Thus, one execution can consist of several periods. **PPlanner** and **CPlanner** create a strategic action for each period of the same strategy execution. Each of these actions contains the initial task to which the strategy was applied in the **task** slot. In its **action-sequence** slot each action contains only those actions that were introduced during the corresponding execution period. Note that the information stored in the strategic actions is not sufficient to identify actions that belong to the same strategy execution. For that purpose also information stored in the corresponding history entries is needed (see section 5.4 for details on the history entries).

PPlanner Action	
strategy	NormalizeLineTask
task	$L_{Thm}.\ L_{Ass_1}, L_{Ass_2} \vdash \exists x.(0 < x \wedge F[x])\ (\mathit{open}) \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$
binding store	$\mathcal{B}S$
action-sequence	$[A_{\exists\text{I-B}}, A_{\wedge\text{I-B}}, \ldots]$

Figure 18: A strategic action of **PPlanner**.

An example for an action of **PPlanner** is given in Figure 18. The strategic action results from the application of the strategy NormalizeLineTask to the line-task
$$L_{Thm}.\ L_{Ass_1}, L_{Ass_2} \vdash \exists x.(0 < x \wedge F[x])\ (\mathit{open}) \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}.$$
First, **PPlanner** applies the method $\exists$I-B to the initial task. Then, it applies the method $\wedge$I-B to the resulting task with task-formula $0 < mv_x \wedge F[mv_x]$. If $F[mv_x]$ is again a complex formula, then **PPlanner** can perform further actions in order to decompose $F[mv_x]$. The sequence of actions performed by **PPlanner**, $[A_{\exists\text{I-B}}, A_{\wedge\text{I-B}}, \ldots]$, is stored in the slot **action-sequence** of the strategic action.

ATP

The algorithm **ATP** employs external automated theorem provers to prove line-tasks. If the automated theorem prover succeeds, then the **ATP** algorithm closes the goal of the line-task and creates a strategic action and stores the output of the external system in the slot **output**.

An example for an action of **ATP** is given in Figure 19. The strategy CallTramp is applied to the (trivial) problem to show that $P \Rightarrow P$ holds. The problem is passed to TRAMP, which provides as output the ND-proof given in the **output** slot of the action.

ATP Action	
strategy	CallTramp
task	$L.\emptyset \vdash P \Rightarrow P \; (open) \blacktriangleleft \emptyset$
binding store	$\mathcal{B}$S
output	$L_1. \quad L_1 \quad \vdash P \qquad (Hyp)$ $L_2. \quad L_1 \quad \vdash P \qquad (Weaken)$ $L. \quad \emptyset \quad \vdash P \Rightarrow P \qquad (\Rightarrow_I L_2)$

Figure 19: A strategic action of **ATP**.

EXP

The algorithm **EXP** expands complex steps, i.e., method or tactic steps in the constructed $\mathcal{PDS}$. For a proof line L with justification $(J \; P_1 \; \dots P_n)$, where J is a method or a tactic and $P_1, \dots, P_n$ are the premises, **EXP** computes a proof segment that derives the conclusion L of the step from its premises $P_1, \dots, P_n$ at a lower level of abstraction. This proof segment is stored in the slot **expansion-segment** of an action of **EXP**. Moreover, an **EXP** action contains the slot **open-lines**, which contains the set of new open lines that are introduced in the expansion-segment.[23]

An example is given in Figure 20. This **EXP** action results from the expansion of the justification (=Subst-B $L_{Thm'}$ L_{Ass_1}) of proof line L_{Thm} (compare with example 2.1 in section 2.4). When this step is expanded, then the proof schema of the method =Subst-B (see section 2.3) is instantiated in order to derive L_{Thm} from the premises $L_{Thm'}$ and L_{Ass_1} as given in the **expansion-segment** in Figure 20.

EXP Action	
strategy	**EXP**
task	$L_{Thm}. \; L_{Ass_1}, L_{Ass_2} \vdash even(a+b)$ (=Subst-B $L_{Thm'} \; L_{Ass_1})\vert^{Exp}$
binding store	$\mathcal{B}$S
expansion-segment	$L_{Ass_1}. \quad L_{Ass_1} \quad \vdash a \doteq c \qquad (Hyp)$ $L_{Thm'}. \quad L_{Ass_1}, L_{Ass_2} \vdash even(c+b) \qquad (Open)$ $L_1. \quad L_{Ass_1}, L_{Ass_2} \vdash \forall P. P(c) \Rightarrow P(a) \qquad (\equiv_E L_{Ass_1} \; (\doteq))$ $L_2. \quad L_{Ass_1}, L_{Ass_2} \vdash (\lambda x. even(x+b))(c) \Rightarrow (\lambda x. even(x+b))(a) \qquad (\forall_E L_1 \; (\lambda x. even(x+b)))$ $L_3. \quad L_{Ass_1}, L_{Ass_2} \vdash even(c+b) \Rightarrow even(a+b) \qquad (\lambda\leftrightarrow L_2)$ $L_{Thm}. \quad L_{Ass_1}, L_{Ass_2} \vdash even(a+b) \qquad (\Rightarrow_E \; L_3 \; L_{Thm'})$
open-lines	$\{\}$

Figure 20: A strategic action of **EXP**.

INSTMETA

The algorithm **INSTMETA** computes instantiations of meta-variables. Actions of **INSTMETA** store the computed instantiation in the slot **instantiation**. An example for an action of **INSTMETA** is

[23] If one of the premises $P_1, \dots, P_n$ is open, then it is not in this slot, since it was not changed by the expansion (i.e., its open justification was not created by the expansion).

given in Figure 21. This action results from the application of the strategy ComputeInstFromCS to the task $mv_\delta|^{Inst}$. **INSTMETA** computes the instantiation $min(c_{\delta_1}, c_{\delta_2})$ for mv_δ and stores it in the **instantiation** slot.

INSTMETA Action	
strategy	ComputeInstFromCS
task	$mv_\delta\|^{Inst}$
binding store	$\mathcal{B}$S
instantiation	$min(c_{\delta_1}, c_{\delta_2})$

Figure 21: A strategic action of **INSTMETA**.

5.3 Strategic Proof Plans

In this section, we shall extend the notions introduced in section 3.1 to strategic proof plans. We start with the definitions of a strategic proof planning problem, an initial $\mathcal{PDS}$ of a strategic proof planning problem (which is the same as the initial $\mathcal{PDS}$ of a proof planning problem), and an initial agenda of a strategic proof planning problem (which is different from the initial agenda of a proof planning problem since it may contains instantiation-tasks).

Definition
A *strategic proof planning problem* is a quadruple $(Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{S}, \mathcal{C}_\mathcal{S})$, where Thm and $Ass_1, \ldots, Ass_n$ are formulas in ΩMEGA's higher-order language, $\mathcal{S}$ is a set of strategies, and $\mathcal{C}_\mathcal{S}$ is a set of strategic control rules. Thm is also called the *theorem* of the strategic proof planning problem whereas $Ass_1, \ldots, Ass_n$ are called the *assumptions* of the strategic proof planning problem. □

Definition
Let $(Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{S}, \mathcal{C}_\mathcal{S})$ be a strategic proof planning problem. The *initial* $\mathcal{PDS}$ of this problem is the $\mathcal{PDS}$ that consists of an open line L_{Thm} with formula Thm and the lines L_{Ass_i} with formula Ass_i and the hypothesis justification *Hyp*, respectively. The *initial agenda* of the strategic proof planning problem is the agenda that consists of the line-task $L_{Thm} \blacktriangleleft \{L_{Ass_1}, \ldots, L_{Ass_n}\}$ and an instantiation-task $mv|^{Inst}$ for each meta-variable in $L_{Thm}, L_{Ass_1}, \ldots, L_{Ass_n}$. □

Next, we extend the action applicability notion of PLAN. In MULTI, actions are applicable with respect to a $\mathcal{PDS}$ and a binding store. In particular, an action is applicable only if the current binding store equals[24] the binding store with respect to which the action was computed (i.e., the binding store that is stored in the slot **binding store** of the action). This restriction is necessary since the computation of actions can rely on given bindings in the current binding store. Moreover, we extend the action introduction functions Φ and $\vec{\Phi}$ of PLAN (see definition 3.1 and definition 3.1) to the strategic action introduction functions Φ_{MULTI} and $\vec{\Phi}_{\text{MULTI}}$. Φ_{MULTI} describes the operational semantics of an action in MULTI when it is applied to an agenda, a $\mathcal{PDS}$, a sequence of actions, and a sequence of bindings stores, i.e., Φ_{MULTI} defines a transition relation between quadruples of agendas, $\mathcal{PDS}$s, sequences of actions, and sequences of binding stores. First, we give general

[24] Two binding stores are equal when they contain the same bindings.

definitions of Φ_{MULTI} and $\vec{\Phi}_{\text{MULTI}}$. Then, we define for each kind of action used in MULTI when it is applicable and the results of its introduction by Φ_{MULTI}.

Definition The *action introduction function* Φ_{MULTI} is a partial function that maps a sequence of actions, an agenda, a $\mathcal{PDS}$, a sequence of binding stores, and an applicable action into a sequence of actions, an agenda, a $\mathcal{PDS}$, and a sequence of binding stores, i.e.,

$$\Phi_{\text{MULTI}} : \vec{A} \times \hat{\text{A}} \times \mathcal{P} \times \vec{\mathcal{B}}\text{S} \times A_{add} \mapsto \vec{A}\text{'} \times \hat{\text{A}}\text{'} \times \mathcal{P}\text{'} \times \vec{\mathcal{B}}\text{S'}.$$

The *recursive action introduction function* $\vec{\Phi}_{\text{MULTI}}$ is a partial function that maps a sequence of actions, an agenda, a $\mathcal{PDS}$, a sequence of binding stores, and a sequence of actions into a sequence of actions, an agenda, a $\mathcal{PDS}$, and a sequence of binding stores, i.e.,

$$\vec{\Phi}_{\text{MULTI}} : \vec{A} \times \hat{\text{A}} \times \mathcal{P} \times \vec{\mathcal{B}}\text{S} \times \vec{A}_{add} \mapsto \vec{A}\text{'} \times \hat{\text{A}}\text{'} \times \mathcal{P}\text{'} \times \vec{\mathcal{B}}\text{S'}.$$

$\vec{\Phi}_{\text{MULTI}}$ is recursively defined as follows:
Let $\vec{A}$ be a sequence of actions, $\hat{\text{A}}$ an agenda, $\mathcal{P}$ a $\mathcal{PDS}$, $\vec{\mathcal{B}}\text{S}$ a sequence of binding stores, and $\vec{A}_{add}$ a sequence of actions.

1. If $\vec{A}_{add}$ is empty, then
   $\vec{\Phi}_{\text{MULTI}}(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, \vec{A}_{add}) := (\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S})$.

2. Otherwise let $A_{add} := \textsf{first}(\vec{A}_{add})$ and $\vec{A}\text{'}_{add} := \textsf{rest}(\vec{A}_{add})$. If A_{add} is applicable with respect to $\mathcal{P}$ and the last binding store of $\vec{\mathcal{B}}\text{S}$, and if $\hat{\text{A}}$ contains the task of A_{add}, then
   $\vec{\Phi}_{\text{MULTI}}(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, \vec{A}_{add}) := \vec{\Phi}_{\text{MULTI}}(\Phi_{\text{MULTI}}(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, A_{add}), \vec{A}\text{'}_{add})$.

□

Method Actions

A method action is applicable with respect to a $\mathcal{PDS}$, if the given lines of the action are in the $\mathcal{PDS}$. Φ_{MULTI} differs from Φ in two points. First, Φ_{MULTI} creates not only new line-tasks but also new instantiation-tasks (for each new meta-variable in the new outlines created by the method action) and new expansion-tasks (for each conclusion of the method action). Second, MULTI allows method actions that contain binding constraints in their **constraints** slot. These binding constraints are labeled with *Binding*, which indicates that they are not passed to an external constraint solver but to the binding store.[25] When the action is introduced, a new binding store is created and added to the sequence of binding stores. The new binding store results from the union of the bindings of the last binding store and the new bindings. The instantiation-tasks whose meta-variables are bound by the new bindings are then removed from the agenda.

Definition Let $\mathcal{P}$ be a $\mathcal{PDS}$, $\mathcal{B}\text{S}$ a binding store, and A_{add} a method action with the binding store $\mathcal{B}\text{S}_{A_{add}}$. Moreover, let $\mathcal{L}$ be the set of proof lines of $\mathcal{P}$ and let $\ominus Concs$ be the $\ominus$ conclusions, $\ominus Prems$ the $\ominus$ premises, and $BPrems$ the blank premises of A_{add}. A_{add} is *applicable* with respect to $\mathcal{P}$ and $\mathcal{B}\text{S}$, if

1. $(\ominus Concs \cup \ominus Prems \cup BPrems)$ is a subset of $\mathcal{L}$,

[25] Internal binding constraints in method actions were first introduced by LASSAAD CHEIKHROUHOU in an extension of PLAN for proof planning diagonalization proofs [11].

2. $\mathcal{B}\text{S}_{A_{add}} = \mathcal{B}\text{S}$.

□

Definition Let $\vec{\mathcal{B}\text{S}}$ be a sequence of bindings stores and let $\mathcal{B}\text{S}$ be the last binding store of $\vec{\mathcal{B}\text{S}}$. Let $\vec{A}$ be a sequence of actions and let A_{add} be a method action, which is applicable with respect to a $\mathcal{PDS}$ $\mathcal{P}$ and $\mathcal{B}\text{S}$.

Moreover, let $\oplus Concs$ be the $\oplus$ conclusions, $\ominus Concs$ the $\ominus$ conclusions, $\oplus Prems$ the $\oplus$ premises, $\ominus Prems$ the $\ominus$ premises, and $BPrems$ the blank premises of A_{add}. Let $T = L_{open} \blacktriangleleft \mathcal{SUPPS}_{L_{open}}$ be the task of A_{add} and let σ be the binding constraints of A_{add}.
$Prems$:=$\oplus Prems \cup \ominus Prems \cup BPrems$,
$Concs$:=$\oplus Concs \cup \ominus Concs$
$New\text{-}Lines$:=$\oplus Concs \cup \oplus Prems$
$New\text{-}Supps$:=$(\mathcal{SUPPS}_{L_{open}} \cup \oplus Concs) - \ominus Prems$.
$New\text{-}Line\text{-}Tasks$:=$[L \blacktriangleleft New\text{-}Supps \mid L \in \oplus Prems]$.
$New\text{-}Inst\text{-}Tasks$:=$[mv|^{Inst} \mid mv \in New\text{-}Lines \textit{ and not } mv|^{Inst} \textit{ in } \hat{\text{A}}]$.
$New\text{-}Exp\text{-}Tasks$:=$[C|^{Exp} \mid C \textit{ in } Concs]$.
$New\text{-}Tasks$:=$New\text{-}Line\text{-}Tasks \ \cup \ New\text{-}Inst\text{-}Tasks \ \cup \ New\text{-}Exp\text{-}Tasks$.
$Old\text{-}Inst\text{-}Tasks$:=$[mv|^{Inst} \mid mv\text{:=}^{b} t \in \sigma]$.
$\hat{\text{A}}_{rest}$:=$\hat{\text{A}} - ([\text{T}] \cup Old\text{-}Inst\text{-}Tasks))$.

If $\hat{\text{A}}$ is an agenda that contains the task T of A_{add}, then the result $(\vec{A}\text{'}, \hat{\text{A}}\text{'}, \mathcal{P}\text{'}, \vec{\mathcal{B}\text{S}}\text{'})$ of $\Phi_{\text{MULTI}}(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}\text{S}}, A_{add})$ is defined by:

- $\vec{A}$':= $\vec{A} \cup [A_{add}]$.
- $\hat{\text{A}}$':= $\begin{cases} New\text{-}Tasks \cup \hat{\text{A}}_{rest} & \text{if } L_{open} \in \ominus Concs, \\ [L_{open} \blacktriangleleft New\text{-}Supps] \cup New\text{-}Tasks \cup \hat{\text{A}}_{rest} & \text{else.} \end{cases}$
- $\mathcal{P}$' results from $\mathcal{P}$ by
  1. adding the proof lines $New\text{-}Lines$, respectively, and
  2. justifying the proof lines $\ominus Concs$ and $\oplus Concs$ by the application of the method of A_{add} to $Prems$, respectively.
- If σ is empty, then $\vec{\mathcal{B}\text{S}}$':=$\vec{\mathcal{B}\text{S}}$. Otherwise, $\vec{\mathcal{B}\text{S}}$':=$\vec{\mathcal{B}\text{S}} \cup [\mathcal{B}\text{S}_{new}]$ where $\mathcal{B}\text{S}_{new} := \{mv_i\text{:=}^{b}\, t_i\sigma | (mv_i\text{:=}^{b}\, t_i) \in \mathcal{B}\text{S}\} \cup \sigma$.[26]

□

[26] $t_i\sigma$ is the term that results from the application of the binding constraints in σ to the subterms of t_i. That is, each occurrence of a meta-variable mv' in t_i that is bound by a constraint $mv\text{:=}^{b}\, t'$ in σ is replaced by an occurrence of t'.

INSTMETA Actions

An **INSTMETA** action is applicable with respect to a binding store and a $\mathcal{PDS}$, if the proof lines of the $\mathcal{PDS}$ contain occurrences of its meta-variable but there is no binding for the meta-variable in the binding store. When applied to an action of **INSTMETA**, Φ_{MULTI} creates a new binding store, which is added to the sequence of binding stores. The new binding store results from adding a binding for the meta-variable of the instantiation-task of the action to the last binding store of the sequence.

Definition Let $\mathcal{P}$ be a $\mathcal{PDS}$ with proof lines $\mathcal{L}$, $\mathcal{B}$S a binding store, and A_{add} an **INSTMETA** action. Let $T_{A_{add}} = mv|^{Inst}$ be the task of A_{add} and $\mathcal{B}\text{S}_{A_{add}}$ its binding store. A_{add} is *applicable* with respect to $\mathcal{P}$ and $\mathcal{B}$S, if

1. there are occurrences of mv in the formulas of the proof lines $\mathcal{L}$,
2. there is no binding for mv in $\mathcal{B}$S,
3. $\mathcal{B}\text{S}_{A_{add}} = \mathcal{B}\text{S}$.

□

Definition Let $\vec{\mathcal{B}}\text{S}$ be a sequence of bindings stores and let $\mathcal{B}$S be the last binding store of $\vec{\mathcal{B}}\text{S}$. Let $\vec{A}$ be a sequence of actions and let A_{add} be an **INSTMETA** action, which is applicable with respect to a $\mathcal{PDS}$ $\mathcal{P}$ and $\mathcal{B}$S.

Moreover, let $T = mv|^{Inst}$ be the task of A_{add} and let t be the instantiation for mv in A_{add}.
σ:=$\{mv{:=}^b t\}$.

If Â is an agenda that contains the task T of A_{add}, then the result ($\vec{A}$', Â', $\mathcal{P}$', $\vec{\mathcal{B}}\text{S}$') of $\Phi(\vec{A}, \text{Â}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, A_{add})$ is defined by:

- $\vec{A}$':= $\vec{A} \cup [A_{add}]$.
- Â':= Â - $[T]$.
- $\mathcal{P}$':= $\mathcal{P}$.
- $\vec{\mathcal{B}}\text{S}$':=$\vec{\mathcal{B}}\text{S} \cup [\mathcal{B}\text{S}_{new}]$ where $\mathcal{B}\text{S}_{new} := \{mv_i{:=}^b t_i\sigma | (mv_i{:=}^b t_i) \in \mathcal{B}\text{S}\} \cup \sigma$.

□

ATP Actions

An **ATP** action is applicable wrt. a $\mathcal{PDS}$, if the proof lines of the line-task of the action are in the $\mathcal{PDS}$. When applied to an action of **ATP** with task $L_{open} \blacktriangleleft \{S_1, \ldots, S_n\}$, Φ_{MULTI} closes L_{open} in the $\mathcal{PDS}$ with an application of the tactic atp. The only resulting new task is an expansion-task for L_{open}.

Definition Let $\mathcal{P}$ be a $\mathcal{PDS}$ with the proof lines $\mathcal{L}$, $\mathcal{B}$S a binding store, and A_{add} an **ATP** action. Let $T_{A_{add}} = L_{open} \blacktriangleleft \{S_1, \ldots, S_n\}$ be the task of A_{add} and $\mathcal{B}\text{S}_{A_{add}}$ its binding store. A_{add} is *applicable* with respect to $\mathcal{P}$ and $\mathcal{B}$S, if

1. $L_{open} \in \mathcal{L}$ and $\mathcal{SUPPS}_{L_{open}} \subseteq \mathcal{L}$,

2. $\mathcal{B}\mathrm{S}_{A_{add}} = \mathcal{B}\mathrm{S}$.

□

Definition Let $\vec{\mathcal{B}}\mathrm{S}$ be a sequence of bindings stores and let $\mathcal{B}$S be the last binding store of $\vec{\mathcal{B}}\mathrm{S}$. Let $\vec{A}$ be a sequence of actions and let A_{add} be an **ATP** action, which is applicable with respect to a $\mathcal{PDS}$ $\mathcal{P}$ and $\mathcal{B}$S.

Moreover, let $T = L_{open} \blacktriangleleft \mathcal{SUPPS}_{L_{open}}$ be the task of A_{add} and let Out be the content of the slot **output** of A_{add}.

If Â is an agenda that contains the task T of A_{add}, then the result $(\vec{A}’, \hat{\mathrm{A}}’, \mathcal{P}’, \vec{\mathcal{B}}\mathrm{S}’)$ of $\Phi(\vec{A}, \hat{\mathrm{A}}, \mathcal{P}, \vec{\mathcal{B}}\mathrm{S}, A_{add})$ is defined by:

- $\vec{A}$’:= $\vec{A} \cup [A_{add}]$.
- Â’:= $(\hat{\mathrm{A}} - [T]) \cup [L_{open}|^{Exp}]$.
- $\mathcal{P}$’ results from $\mathcal{P}$ by justifying the proof line L_{open} with an application of the tactic atp to the supports $\mathcal{SUPPS}_{L_{open}}$ and the parameter Out.
- $\vec{\mathcal{B}}\mathrm{S}$’:=$\vec{\mathcal{B}}\mathrm{S}$.

□

EXP Actions

An **EXP** action is applicable with respect to a $\mathcal{PDS}$, if the closed line in the expansion-task of the action is in the $\mathcal{PDS}$ and if the premises of the justification of the closed line are in the $\mathcal{PDS}$. When applied to an action of **EXP**, Φ_{MULTI} introduces the new proof lines of the **expansion-segment** slot into the $\mathcal{PDS}$ and adds all resulting new tasks to the agenda, namely new instantiation-tasks for new meta-variables in the new proof lines, new line-tasks for open lines in the new proof lines, and new expansion-tasks for all new proof lines, which have a tactic or a method justification.

Definition Let $\mathcal{P}$ be a $\mathcal{PDS}$ with the proof lines $\mathcal{L}$, $\mathcal{B}$S a binding store, and A_{add} an **EXP** action with the binding store $\mathcal{B}\mathrm{S}_{A_{add}}$. Moreover, let $T_{A_{add}} = L|^{Exp}$ be the task of A_{add} where L has the justification $(J\ P_1\ \ldots P_n)$. A_{add} is *applicable* with respect to $\mathcal{P}$ and $\mathcal{B}$S, if

1. $L \in \mathcal{L}$ and $\{P_1\ \ldots P_n\} \subseteq \mathcal{L}$,

2. $\mathcal{B}\mathrm{S}_{A_{add}} = \mathcal{B}\mathrm{S}$.

□

Definition Let $\vec{\mathcal{B}}\mathrm{S}$ be a sequence of bindings stores and let $\mathcal{B}$S be the last binding store of $\vec{\mathcal{B}}\mathrm{S}$. Let $\vec{A}$ be a sequence of actions and let A_{add} be an **EXP** action, which is applicable with respect to a $\mathcal{PDS}$ $\mathcal{P}$ and $\mathcal{B}$S.

Moreover, let $T = L|^{Exp}$ be the task of A_{add} and $(J\ P_1 \ldots P_n)$ the justification of L (before the expansion).
$\mathcal{SUPPS}$:=$\{P_1, \ldots, P_n\}$.
$New\text{-}Lines$:=**expansion-segment** of A_{add} without $L, P_1, \ldots, P_n$.
$New\text{-}Open\text{-}Lines$:=**open-lines** of A_{add}.
$New\text{-}Line\text{-}Tasks$:=$[L' \blacktriangleleft \mathcal{SUPPS} \mid L'\ in\ New\text{-}Open\text{-}Lines]$.
$New\text{-}Inst\text{-}Tasks$:=$[mv|^{Inst} \mid mv \in New\text{-}Lines\ and\ not\ mv|^{Inst}\ in$ Â].
$New\text{-}Exp\text{-}Tasks$:=$[L'|^{Exp} \mid (L' \in New\text{-}Lines\ or\ L' = L)\ andL'\ closed\ by\ tactic\ or\ method]$.
$New\text{-}Tasks$:=$New\text{-}Line\text{-}Tasks \cup New\text{-}Inst\text{-}Tasks \cup New\text{-}Exp\text{-}Tasks$.

If Â is an agenda that contains the task T of A_{add}, then the result $(\vec{A}$', Â', $\mathcal{P}$', $\vec{\mathcal{B}}$S') of $\Phi(\vec{A}$, Â, $\mathcal{P}, \vec{\mathcal{B}}$S, $A_{add})$ is:

- $\vec{A}$':= $\vec{A} \cup [A_{add}]$.
- Â':= (Â − [T]) $\cup$ $New\text{-}Tasks$.
- $\mathcal{P}$' results from $\mathcal{P}$ by
  1. adding the new justification specified in the expansion segment to L as the justification of the lowest level of abstraction, and
  2. adding the proof lines $New\text{-}Lines$.
- $\vec{\mathcal{B}}$S':=$\vec{\mathcal{B}}$S.

□

PPLANNER and CPLANNER Actions

A **PPLANNER** or **CPLANNER** action A_S is applicable, if all actions $[A_1, \ldots, A_n]$ in its **action-sequence** slot are applicable when introduced successively. When applied to A_S, Φ_{MULTI} stepwise introduces the actions from the sequence $[A_1, \ldots, A_n]$ using the function $\vec{\Phi}_{\text{MULTI}}$. Afterwards, it replaces $[A_1, \ldots A_n]$ in the constructed action sequence by A_S. That is, the actions $A_1, \ldots, A_n$ are not explicitly mentioned in the constructed action sequence but only implicitly as part of the action of **PPLANNER** or **CPLANNER**. This guarantees that Φ_{MULTI} and $\vec{\Phi}_{\text{MULTI}}$ create a sequence of strategic actions.

Definition Let $\mathcal{P}$ be a $\mathcal{PDS}$, $\mathcal{B}$S a binding store, and A_{add} a **PPLANNER** or **CPLANNER** with the action sequence $[A_1, \ldots, A_n]$. Moreover, let $T_{A_{add}}$ be the task of A_{add} and $\mathcal{B}S_{A_{add}}$ its binding store. A_{add} is *applicable* with respect to $\mathcal{P}$ and $\mathcal{B}$S, if for each $A_i, i = 1 \ldots n$ in $[A_1, \ldots, A_n]$ holds:

- Let $(\vec{A}_i$, Â$_i, \mathcal{P}_i, \vec{\mathcal{B}}S_i) := \vec{\Phi}_{\text{MULTI}}(\vec{A}$, Â, $\mathcal{P}, \vec{\mathcal{B}}$S, $[A_1, \ldots, A_{i-1}])$ for an arbitrary sequence of actions $\vec{A}$ and an agenda Â that contains the task $T_{A_{add}}$. Then, A_i is applicable with respect to $\mathcal{P}_i$, and $\vec{\mathcal{B}}$S$_i$ and Â$_i$ contains the task of A_i.

□

Definition Let $\vec{\mathcal{B}}$S be a sequence of bindings stores and let $\mathcal{B}$S be the last binding store of $\vec{\mathcal{B}}$S. Let $\vec{A}$ be a sequence of actions and let A_{add} be a **PPLANNER** or **CPLANNER** action, which is applicable with respect to a $\mathcal{PDS}$ $\mathcal{P}$ and $\mathcal{B}$S.

Moreover, let $[A_1, \ldots, A_n]$ be the action-sequence of A_{add}.
$(\vec{A}_{rec}, \hat{\text{A}}_{rec}, \mathcal{P}_{rec}, \vec{\mathcal{B}}\text{S}_{rec}) := \vec{\Phi}_{\text{MULTI}}(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, [A_1, \ldots, A_n])$.

If Â is an agenda that contains the task of A_{add}, then the result $(\vec{A}\text{'}, \hat{\text{A}}\text{'}, \mathcal{P}\text{'}, \vec{\mathcal{B}}\text{S'})$ of $\Phi(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, A_{add})$ is defined by:

- $\vec{A}$':= $(\vec{A}_{rec} - [A_1, \ldots, A_n]) \cup [A_{add}]$.
- Â':= $\hat{\text{A}}_{rec}$.
- $\mathcal{P}$':= $\mathcal{P}_{rec}$.
- $\vec{\mathcal{B}}$S':=$\vec{\mathcal{B}}\text{S}_{rec}$.

□

With the function $\vec{\Phi}_{\text{MULTI}}$ we can define strategic proof plans and strategic solution proof plans. Actually, we shall give three different notions of solution proof plans, which specify more and more strict conditions for strategic proof plans.

Definition
Let $(Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{S}, \mathcal{C}_{\mathcal{S}})$ be a strategic proof planning problem, $\mathcal{P}_{init}$ the initial $\mathcal{PDS}$ of this problem, and $\hat{\text{A}}_{init}$ its initial agenda.

A *strategic proof plan* to the strategic proof planning problem is a quadruple SPP $= (\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S})$ with a sequence of strategic actions $\vec{A}$, an agenda Â, a $\mathcal{PDS}$ $\mathcal{P}$, and a sequence of binding stores $\vec{\mathcal{B}}$S such that:

1. each strategy of an action of $\vec{A}$ is in $\mathcal{S}$,
2. $(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S}) = \vec{\Phi}_{\text{MULTI}}([], \hat{\text{A}}_{init}, \mathcal{P}_{init}, [], \vec{A})$,

□

With respect to this definition of a strategic proof plan we can also say that Φ_{MULTI} maps a strategic proof plan and an action into a strategic proof plan and that $\vec{\Phi}_{\text{MULTI}}$ maps a strategic proof plan and a sequence of strategic actions into a strategic proof plan.

Definition
Let $(Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{S}, \mathcal{C}_{\mathcal{S}})$ be a strategic proof planning problem, $\mathcal{P}_{init}$ the initial $\mathcal{PDS}$ of this problem, and $\hat{\text{A}}_{init}$ its initial agenda.

We distinguish the following three notions of a *strategic solution proof plan*:

- A *method-level solution proof plan* for the problem is a sequence of strategic actions $\vec{A}$ such that $\vec{\Phi}_{\text{MULTI}}([], \hat{\text{A}}_{init}, \mathcal{P}_{init}, [], \vec{A})$ results in an agenda without line-tasks and a closed $\mathcal{PDS}$.

- An *instantiated method-level solution proof plan* for the problem is a sequence of strategic actions $\vec{A}$ such that $\vec{\Phi}_{\textsc{Multi}}([], \hat{\mathrm{A}}_{init}, \mathcal{P}_{init}, [], \vec{A})$ results in an agenda without line-tasks and instantiation-tasks, a closed $\mathcal{PDS}$, and a binding store sequence such that the last binding store contains bindings for all meta-variables occurring in proof lines of the final $\mathcal{PDS}$.

- A *full solution proof plan* for the problem is a sequence of strategic actions $\vec{A}$ such that $\vec{\Phi}_{\textsc{Multi}}([], \hat{\mathrm{A}}_{init}, \mathcal{P}_{init}, [], \vec{A})$ results in an empty agenda, a closed $\mathcal{PDS}$ in which all nodes are justified by ND-rules, and a binding store sequence such that the last binding store contains bindings for all meta-variables occurring in proof lines of the final $\mathcal{PDS}$.

□

The first notion of solution proof plan is called *method-level solution proof plan* since a strategic proof plan satisfying these conditions is reached by computing method actions whose introduction satisfies all line-tasks and creates a closed $\mathcal{PDS}$. Instantiation-tasks and expansion-tasks can be ignored. The second notion of solution proof plan, *instantiated method-level solution proof plan*, demands to tackle also instantiation-tasks. However, expansion-tasks can still be ignored. Finally, in order to obtain a *full solution proof plan* the expansion-tasks have to be solved. We shall describe in section 5.6.2 how a user can make Multi search for a particular kind of solution proof plan.

5.4 Strategic Manipulation Records

Similar to PLAN, Multi constructs a history consisting of *manipulation records*. These manipulation records contain information, which can be used by the control rules in order to perform meta-reasoning.

Strategy-Application:	
agenda	
alternative-job-offers	
introduced-action	
new-tasks	
execution-message	

Figure 22: A strategy-application record.

A strategy execution of the algorithms **Exp**, **ATP**, and **InstMeta** creates one so-called *strategy-application record* (see Figure 22). The slots **agenda** and **alternative-job-offers** capture the context in which the manipulation was done whereas the the slots **introduced-action**, **new-tasks**, and **execution-message** store the result of the manipulation. The slot **agenda** captures the agenda before the strategy is applied. The slot **alternative-job-offers** contains the list of alternative job offers, when the strategy was applied. The first job offer in this list is the applied strategy and the task to which the strategy was applied. The performed manipulation, namely the action introduced by the execution of the strategy, is stored in the **introduced-action** slot. This slot is empty, if the execution of a strategy failed. The new tasks created by the introduction of the action are stored in the slot **new-tasks**. The slot **execution-message** contains the execution-message returned by the strategy execution.

Strategy executions of the algorithms PPlanner and CPlanner create two manipulation records. When they are invoked or re-invoked, they create a *strategy-start record*; when they terminate or are interrupted, then they create a *strategy-stop record*. Figure 23 shows the skeletons of these two manipulation records.

Strategy-Start:	
agenda	
alternative-job-offers	
task-tag	

Strategy-Stop:	
task-tag	
introduced-action	
new-tasks	
execution-message	

Figure 23: Manipulation records created by PPlanner and CPlanner.

The strategy-start and strategy-stop records divide the information of a strategy-appli-cation record into two parts: the information available when the strategy is invoked or re-invoked, which is stored in a strategy-start record, and the information available when the strategy stops, which is stored in a strategy-stop record. Hence, a strategy-start record has the slots **agenda** and **alternative-job-offers** whereas a strategy-stop record has the slots **introduced-action**, **new-tasks**, and **execution-message**. Additionally, both records have the slot **task-tag**, which contains the task-tag that uniquely identifies the strategy execution.

Note that the manipulation records of the steps performed within a strategy execution of PPlanner or CPlanner are themselves part of the history. They are not stored in a PPlanner or CPlanner history element but only delimited by the strategy-start and strategy-stop records of the strategy execution. This approach makes information available as early as possible. In particular, the information on the situation when the strategy was invoked or re-invoked and the information on all steps performed by a strategy execution so far are available for the control rules evaluated within the strategy execution.

Strategies of the BackTrack algorithm create two manipulation records whose skeletons are given in Figure 24. The *backtrack-start record* contains the information available when the backtracking is started (stored in the **agenda** and **alternative-job-offers** slots) as well as the information which actions the strategy decided to delete. The *backtrack-stop record* contains the information available when the BackTrack strategy stops. Since strategies of BackTrack do not create actions, this record contains only a slot for the execution message.

BackTrack-Start:	
agenda	
alternative-job-offers	
actions-to-delete	

BackTrack-Stop:	
execution-message	

Figure 24: Manipulation records created by BackTrack.

Similar to CPlanner and PPlanner, strategy executions of BackTrack successively perform also a set of individual steps. When executed, a strategy of BackTrack computes first which actions it has to delete. These actions are stored in the start record. However, in order to delete these actions maybe other actions have to be deleted as well (see section 5.5.7 for details). All single deletion steps are stored in action-deletion records as in PLAN (see section 3). Hence, a start and stop record pair of a **BackTrack** strategy execution delimits the manipulation records of all single deletion steps performed within this strategy execution.

5.5 The Algorithms

In this section, we shall describe the algorithms used in MULTI. First, we explain MULTI's top-level algorithm. Then, we describe the refinement and modification algorithms integrated so far, namely **PPlanner**, **CPlanner**, **Exp**, **ATP**, **InstMeta**, and **BackTrack**.

In the remainder of this section we assume that each function and algorithm used in MULTI has access to the blackboards and the entries on them. Hence, when an algorithm or a function accesses information from a blackboard we shall not mention the respective blackboard explicitly as an argument of the function. The only exceptions are the functions *write-onto-blackboard*, which sets the value of an entry on a blackboard, and *take-from-blackboard*, which returns the value of an entry on a blackboard. Both functions obtain the blackboard on which they should work as argument. In the following descriptions of the algorithms we use PB and CB as abbreviations for the proof blackboard and the control-blackboard, respectively.

5.5.1 The MULTI Algorithm

Figure 25 gives a pseudo-code description of the MULTI algorithm. MULTI is applied to a strategic proof planning problem with a theorem Thm, a set of assumptions $Ass_1, \ldots, Ass_n$, a set of strategies $\mathcal{S}$, and a set of strategic control rules $\mathcal{C}_\mathcal{S}$. Its output is a strategic proof plan for the given problem $(Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{S}, \mathcal{C}_\mathcal{S})$. MULTI's first step is to initialize the proof and the control blackboard. It writes onto the proof blackboard an empty sequence of actions, the initial agenda and the initial $\mathcal{PDS}$ of the given problem, and a sequence of binding stores whose only entry consists of an empty binding store. Moreover, it writes onto the control blackboard an empty set of memory entries, an empty set of demands, and an empty sequence of job offers.

Input: A strategic proof planning problem $(Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{S}, \mathcal{C}_\mathcal{S})$ with a theorem formula Thm, a set of assumption formulas $Ass_1, \ldots, Ass_n$, a list of strategies $\mathcal{S}$, and a list of strategic control rules $\mathcal{C}_\mathcal{S}$.

Output: A strategic proof plan SPP $= (\vec{A}$, Â, $\mathcal{P}$, $\vec{\mathcal{B}}$S) with a sequence of strategic actions $\vec{A}$, an agenda Â, a $\mathcal{PDS}$ $\mathcal{P}$, and a sequence of binding stores $\vec{\mathcal{B}}$S.

```
Algorithm: MULTI($Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{S}, \mathcal{C}_\mathcal{S}$)

  1. Initialization
     Let Â:=initial-agenda($Thm, \{Ass_1, \ldots, Ass_n\}$).
     Let $\mathcal{P}$:=initial-$\mathcal{PDS}$($Thm, \{Ass_1, \ldots, Ass_n\}$).
     write-onto-blackboard([], sequence-of-actions, $PB$).
     write-onto-blackboard(Â, agenda, $PB$).
     write-onto-blackboard($\mathcal{P}$, pds, $PB$).
     write-onto-blackboard([{ }], sequence-of-binding-stores, $PB$).
     write-onto-blackboard([], history, $PB$).
     write-onto-blackboard($\emptyset$, memory, $CB$).
     write-onto-blackboard($\emptyset$, demands, $CB$).
     write-onto-blackboard([], job-offers, $CB$).
  2. Job Offers
     trigger-jobs-from-strategies($\mathcal{S}$).
     trigger-jobs-from-memory().
  3. Guidance
     invoke(MetaReasoner, $\mathcal{C}_\mathcal{S}$).
  4. Invocation
     Let $\mathcal{J}$:=remove-free-jobs(take-from-blackboard(job-offers, $CB$)).
     If $\mathcal{J} = \emptyset$
       then
         terminate and return
          (take-from-blackboard(sequence-of-actions, $PB$),
           take-from-blackboard(agenda, $PB$),
           take-from-blackboard(pds, $PB$),
           take-from-blackboard(sequence-of-binding-stores, $PB$)).
       else
         Let $J$:=first($\mathcal{J}$).
         If job-offer-from-strategy($J$)
           then (i.e., $J = (S, T)$)
             invoke(algorithm-of-strategy($S$), $(S, T)$, $\mathcal{J}$).
           else (i.e., $J = (@_T, Demands)$)
             invoke(algorithm-of-task-tag($@_T$), $@_T$, $\mathcal{J}$).
  5. Execution
     Wait until strategy-ks-terminated().
  6. Administration
     If strategy-ks-terminated-successful(), then delete-satisfied-demands().
     Goto step 2.
```

Figure 25: The MULTI algorithm.

The next four steps, steps 2—5 in Figure 25, of MULTI perform the strategy selection and invocation cycle that is sketched in Figure 13 in the previous section. Step 2 employs the functions *trigger-jobs-from-strategies* and *trigger-jobs-from-memory*. *trigger-jobs-from-strategies* checks whether the condition of an element of S is satisfied by some tasks of the current agenda on the proof blackboard. A strategy $S \in \mathcal{S}$ places a job offer onto the control blackboard for each task T for which its condition is true. The function *trigger-jobs-from-memory* writes for each memory entry a job offer onto the control blackboard. Afterwards, step 3 invokes the MetaReasoner, which evaluates the strategic control rules $\mathcal{C}_S$ on the job offers.

In step 4, MULTI first reads the resulting list of job offers and deletes the job offers whose strategies have still uninstantiated free parameters. If the resulting list is empty, then MULTI terminates and returns the strategic proof plan (i.e., the sequence of actions, the agenda, the $\mathcal{PDS}$, and the sequence of binding stores) on the proof blackboard. Otherwise MULTI picks the first job offer and invokes the corresponding strategy. If the job offer was placed by a strategy S with respect to a task T, which satisfies the condition of S, then MULTI invokes the algorithm of S with the pair (S, T) as argument. If the job offer was placed from a memory entry with task tag $@_T$, then *algorithm-of-task-tag* computes the algorithm that created the tag $@_T$ using information stored in the history and invokes this algorithm with $@_T$ as argument. In both cases the invoked algorithm obtains the list of all job offers on the control blackboard as second argument.

The invoked algorithm refines or modifies the proof blackboard objects and maybe places demands and a memory entry onto the control blackboard. MULTI waits until the execution of the strategy terminates (see step 5). Then, step 6 checks whether the strategy terminated successfully. This check is performed by the function *strategy-ks-terminated-successful*, which looks up the execution message of the last history on the proof blackboard. If this execution message is a success message, then MULTI employs the function *delete-satisfied-demands* to delete all demands on the control blackboard that are satisfied by the terminated strategy execution as well as all pointers in memory entries to those demands. Afterwards, MULTI restarts its cycle by proceeding with step 2.

We conclude this section with two remarks on the described algorithm:

1. When employing the two functions *trigger-jobs-from-memory* (in step 2) and *delete-satisfied-demands* (in step 6) MULTI changes the content of the control blackboard. This is a violation of the blackboard principle, which states that the content of the blackboards should only be changed by respective knowledge sources. For the sake of simplicity of MULTI's blackboard approach we implemented these minor blackboard changes as direct functionalities of the MULTI algorithm. However, in order to avoid a violation of the blackboard principle, we could understand these two functions as particular knowledge sources working on the control blackboard, which are scheduled by MULTI in a pre-defined way.

2. PLAN terminates either with a solution proof plan or, after traversing the search space, with a failure. MULTI terminates as soon as there is no further job offer to invoke (see step 4). However, the lack of job offers states nothing about the status of the strategic proof planning process. When there are no further tasks in the agenda, then there are no further job offers since there is a strategic solution proof plan on the proof blackboard. But it is possible that there are still tasks in the agenda although there are no further job offers. It is possible that there are no strategies to tackle these tasks (i.e., there is no strategy whose condition is satisfied by the task) or strategic control rules can remove all existing job

offers. If MULTI terminates and there are still tasks in the agenda, then it is up to the user to analyze the situation. Is the strategic proof plan created so far a sufficient solution proof plan (when the user is interested in a method-level solution proof plan then expansion-tasks and instantiation-tasks can be ignored)? Are further strategies needed that can deal with particular tasks? Are less restrictive strategic control rules needed that do not remove so much job offers?

5.5.2 The **PPLANNER** Algorithm

Strategies of the algorithm **PPLANNER** refine a strategic proof plan by successively adding method actions, which **PPLANNER** abstracts in one strategic action, when it terminates. A strategy of **PPLANNER** specifies four parameters: a procedure for the computation of the next method action to introduce, parameters for the set of usable methods and control rules, and a termination condition. We discussed some strategies of **PPLANNER** already in section 4.1.

Figure 26 gives a pseudo-code description of the **PPLANNER** algorithm. **PPLANNER** obtains two arguments. When a **PPLANNER** strategy S is initially invoked, then **PPLANNER**'s first input is a pair (S, T) consisting of the strategy S and a line-task T. When a strategy execution is re-invoked, then the first argument is the task tag of the strategy execution. The second argument for **PPLANNER** is the list of all alternative job offers on the control blackboard, when **PPLANNER** is invoked. **PPLANNER** returns no specific output but updates the content of the proof blackboard by introducing successively method actions. Essentially, **PPLANNER** performs a cycle of task selection, action selection, and action introduction, which is similar to the cycle of PLAN. This core cycle is completed by an initialization step and different events that stop the **PPLANNER** algorithm, namely successful termination, interruption, and failure.

In the initialization step (step 1 in Figure 26) **PPLANNER** extracts the information of the strategy and the initial task with respect to which it runs. First, it employs the function *extract-from-input*, which computes the current task tag $@_T$, the current strategy S, and the initial task T. If the first input of **PPLANNER** is a pair (S, T) (i.e., initial call of S on T), then the information on S and T is directly accessible and *extract-from-input* creates a new task tag $@_T$, which it attaches to T. If the first input of **PPLANNER** is a task tag $@_T$ (i.e., re-invocation of interrupted strategy execution), then *extract-from-input* employs information from the history to compute the strategy S and the initial task T that correspond to the given task tag. Next, **PPLANNER** uses the function *parameters-of-strategy* to obtain the parameters of the strategy S, which are a list of methods $\mathcal{M}$, a list of control rules $\mathcal{C}$, the termination condition, and the action computation and selection procedure. So far, we have implemented two action computation and selection procedures, namely **CHOOSEACTION** (see section 3.4) and **CHOOSEACTIONALL** (see appendix A).[27] Afterwards, **PPLANNER** adds a strategy-start record to the history and sets the algorithm variable $\vec{A}_{add}$ to the empty list. In this variable **PPLANNER** stores the method actions, which it introduces successively.

Step 2 and step 3 in Figure 26 check whether **PPLANNER** terminates successfully or interrupts. We postpone the detailed discussion of these two steps until the discussion of step 7 in order to discuss together all three steps that stop **PPLANNER** and the differences among them. The next

[27] Note that parts of these algorithms work slightly differently when used in MULTI as opposed to the functionality described in section 3.4 and appendix A. All functions used within these algorithms that match proof lines of a method with proof lines of a task (e.g., *match-task-line*, *match-s+p* see section 3.4) apply first the bindings of the current binding store to the proof lines of the task. Then, they perform the respective matchings with respect to this "up-to-date" proof lines instead of the original ones.

Input: (1) either a pair (S, T) where S is a **PPLANNER** strategy and T is a line-task or a task tag $@_T$, (2) the list of all alternative job offers $\mathcal{J}$.

Output: No output, only changes of the blackboards.

```
Algorithm: PPLANNER$(arg_1, \mathcal{J}_{rest}))$

  1. Initialization
     Let $(@_T, S, T)$:=extract-from-input$(arg_1)$.
     Let $(\mathcal{M}, \mathcal{C}, term\text{-}cond, action\text{-}proc)$:=parameters-of-strategy$(S)$.
     add-strategy-start-record-to-history$(\mathcal{J}_{rest}, @_T)$.
     Let $\vec{A}_{add}$:=liste.

  2. Successful Termination Check
     (see Figure 27)

  3. Interruption Check
     (see Figure 27)

  4. Task Selection:
     Let current task $T_{curr}$:= first(evalcrules-tasks(tasks-with-tag$(@_T), \mathcal{C}$)).

  5. Action Selection
     Let ($A_{add}$,$\mathcal{A}$):=apply$(action\text{-}proc, T_{curr}, \mathcal{M}, \mathcal{C})$ where $A_{add}$ is an action and $\mathcal{A}$
     is the set of computed alternative actions.

  6. Action Introduction
     If $A_{add}$ is given
       then
          PB:=$\Phi_{\text{MULTI}}(A_{add}, PB)$.
          add-action-intro-record($A_{add}$,$\mathcal{A}$).
          $\vec{A}_{add}$:=$\vec{A}_{add} \cup [A_{add}]$.
          If extract-constraints$(A_{add}) \neq \emptyset$
            then
               pass-constraints(extract-constraints$(A_{add})$).
          Goto step 2.

  7. Failure
     (see Figure 27)
```

Figure 26: The **PPLANNER** algorithm.

three steps — step 4, step 5, and step 6 — are the core cycle of selecting the next task, computing and selecting the next method action, and introducing the selected action. Essentially, these steps correspond to step 2, step 3, and step 4 of PLAN in Figure 8 in section 3.2, they are only slightly adapted to MULTI. When **PPLANNER** selects the next task to tackle in step 4, then it evaluates the control rules of kind 'Task' not on the whole agenda of the proof blackboard, but only on the tasks that carry the current task tag $@_T$ (the restricted initial alternative list is computed by the function *tasks-with-tag*). Whereas in PLAN the application of the algorithm **CHOOSEACTION** is fix, **PPLANNER** applies the action computation procedure specified as parameter of the current strategy in step 5. When an action is found, then **PPLANNER** applies this action in step 6 with the function Φ_{MULTI} to the action sequence, the agenda, the $\mathcal{PDS}$, and the sequence of binding stores on the proof blackboard. We write this as "PB:=$\Phi_{\text{MULTI}}(A_{add}, PB)$" and do not refer to the changed elements of the proof blackboard explicitly. Similar to PLAN, **PPLANNER** adds a history entry for the introduced action and passes new constraints to external constraint solvers. Additionally, the introduced action is added to $\vec{A}_{add}$. Afterwards, **PPLANNER** continues with step 2.

PPLANNER can stop at three different places, namely step 2, step 3 and step 7, which are given in detail in Figure 27. Step 2 checks whether the application of the strategy of **PPLANNER** was successful such that **PPLANNER** should stop. This is the case either when the termination condition of the strategy is satisfied or when there are no further tasks which carry the task tag of the strategy execution. Step 3 employs the function *evalcrules-interrupt* to evaluate the control rules of kind 'Interrupt' on the alternative list [**False**,**True**], where **False** causes no interrupt whereas **True** causes an interrupt. The control rules of kind 'Interrupt' can also compute demands and attach the demands to the **True** element of the alternative list. Finally, step 7 is performed, when step 5 does not provide a method action to introduce, that is, step 7 deals with a failure situation in **PPLANNER**.

Some computations are the same in all three steps. They all compute an execution message $message$ and employ the function *create-strategic-action* to compute a strategic action A^S_{add} from the collected sequence of method actions $\vec{A}_{add}$. Moreover, they all replace the sequence of method actions by a new strategic action in the action sequence on the proof blackboard (this is done by the function *replace-actions*). Finally, they all add a strategy-stop entry to the history before they terminate. The three steps differ in the created execution message and in whether and which memory entries and demands they create. When the strategy knowledge source terminates successfully, then **PPLANNER** creates a success message and does not write memory entries or demands onto the control blackboard. Rather, it applies the function *remove-tag*, which removes its task tag from all tasks in the agenda on the proof blackboard. If the execution of the strategy interrupts, then it creates an interruption message and places a memory entry and demands onto the control blackboard. The demands stem from the evaluated control rules of kind 'Interrupt' and the memory entry consists of the task tag and pointers to the added demands. If **PPLANNER** has to deal with a failure occurring with respect to the task T_{curr}, then it creates a failure message. Moreover, it writes a task-demand $? - ON - T_{curr}$ and a memory entry consisting of the task tag and a pointer to this task-demand onto the control blackboard. Since a failure creates a memory entry and a demand, we can understand it as a special kind of interrupt — the difference with respect to the origin of the interruption is recorded in the execution messages.

The further interpretation of and reaction to the termination is left to MULTI and meta-reasoning at the strategy-level (this holds also for all other refinement and modification algorithms employed by MULTI, which can terminate in different ways). If the last strategy execution terminated with a success message, then MULTI deletes all demands on the control blackboard that are satisfied by this strategy execution (see previous section). Moreover, strategic control rules can make use of the information contained in the execution messages. For instance, the strategic control rule `prefer-backtrack-if-failure` (see section 4.3) analyzes the execution messages and prefers to perform some backtracking if the last strategy was a **PPLANNER** strategy and terminated with a failure message. This control rule (which can be overwritten by more specific control rules) forces a systematic traversal of the search space given by a **PPLANNER**

```
2. Termination Check
   If no-tasks-with-tag$(@_T)$ or apply$(term\text{-}cond) = true$
     then
       Let $message$:=create-success-message$(S, T)$.
       Let $A^S_{add}$:=create-strategic-action$(\vec{A}_{add})$.
       replace-actions$(\vec{A}_{add}, A^S_{add})$.
       remove-tag$(@_T)$.
       add-strategy-stop-record-to-history$(@_T, A^S_{add}, message)$.
       Terminate.

3. Interruption Check
   Let $I$:=first(evalcrules-interrupt$([\mathbf{Nil}, \mathbf{True}], \mathcal{C})$).
   If $I = \mathbf{True}$
     then
       Let $message$:=create-interrupt-message$(S, T)$.
       Let $A^S_{add}$:=create-strategic-action$(\vec{A}_{add})$.
       replace-actions$(\vec{A}_{add}, A^S_{add})$.
       write-to-demands(demands$(I)$).
       write-to-memory$(@_T$, demands$(I))$.
       add-strategy-stop-record-to-history$(@_T, A^S_{add}, message)$.
       Terminate.

7. Failure
   IF $A_{add}$ is not given
     then
       Let $message$:=create-failure-message$(S, T)$.
       Let $A^S_{add}$:=create-strategic-action$(\vec{A}_{add})$.
       replace-actions$(\vec{A}_{add}, A^S_{add})$.
       write-to-demands$(\{? - ON - T\})$.
       write-to-memory$(@_T, \{? - ON - T\})$.
       add-strategy-stop-record-to-history$(@_T, A^S_{add}, message)$.
       Terminate.
```

Figure 27: Leaving the **PPLANNER** algorithm.

strategy.

5.5.3 The CPLANNER Algorithm

Strategies of the algorithm **CPLANNER** refine a strategic proof plan by successively transferring actions from a source proof plan into the proof plan under construction. A strategy of **CPLANNER** specifies three parameters: a list of action transfer procedures, a list of control rules, and a termination condition. We discussed an example strategy of **CPLANNER** already in section 4.4. More examples are discussed in [55].

Figure 28 gives a pseudo-code description of **CPLANNER**. **CPLANNER** obtains two arguments. When a **CPLANNER** strategy S is initially invoked, then **CPLANNER**'s first input is a pair (S, T) consisting of the strategy S and a line-task T. When a strategy execution is re-invoked, then the first argument is the task tag of the strategy execution. The second argument for **CPLANNER** is the list of all alternative job offers on the control blackboard, when **CPLANNER** is invoked. **CPLANNER** returns no specific output but updates the content of the proof blackboard by introducing successively method actions.

Input: (1) either a pair (S, T) where S is a **CPLANNER** strategy and T is a task or a task tag $@_T$, (2) the list of all alternative job offers $\mathcal{J}$.

Output: No output, only changes of the blackboards.

Algorithm: CPLANNER$(arg_1, \mathcal{J}_{rest}))$

```
1. Initialization
   Let $(@_T, S, T)$:=extract-from-input$(arg_1)$.
   Let $(\mathcal{TP}, \mathcal{C}, term\text{-}cond)$:=parameters-of-strategy$(S)$.
   add-strategy-start-record-to-history$(\mathcal{J}_{rest}, @_T)$.
   Let $\vec{A}_{add}$:=[].

2. Successful Termination Check
   (see PPLANNER Figure 27)

3. Interruption Check
   (see PPLANNER Figure 27)

4. Select and Evaluate Transfer Procedures
   Let $\mathcal{TP}_{rest}$:=evalcrules-transferprocs$(\mathcal{TP})$.
   Until ($Obj$ is action or demand) or ($\mathcal{TP}_{rest} = []$)
         Let $TP_{curr}$:=first$(\mathcal{TP}_{rest})$.
         Let $Obj$:=evaluate$(TP_{curr})$.
         $\mathcal{TP}_{rest}$:=rest$(\mathcal{TP}_{rest})$.

5. Action Introduction
   If $Obj$ is action $A_{add}$
      then
         PB:=$\Phi_{\text{MULTI}}(A_{add}, PB)$.
         add-action-intro-record($A_{add}$,$\mathcal{A}$).
         $\vec{A}_{add}$:=$\vec{A}_{add} \cup [A_{add}]$.
         If extract-constraints$(A_{add}) \neq \emptyset$
            then pass-constraints(extract-constraints$(A_{add})$).
         Goto step 2.

6. Demand Interruption
   If $Obj$ is demand $D_{add}$
      then
         Let $message$:=create-interrupt-message$(S, T)$.
         Let $A^S_{add}$:=create-strategic-action$(\vec{A}_{add})$.
         replace-actions$(\vec{A}_{add}, A^S_{add})$.
         write-to-demands$(D_{add})$.
         write-to-memory$(@_T, D_{add})$.
         add-strategy-stop-record-to-history$(@_T, A^S_{add}, message)$.
         Terminate.

7. Failure
   (see PPLANNER Figure 27)
```

Figure 28: The **CPLANNER** algorithm.

Several parts of the **CPlanner** algorithm are equal or similar to the **PPlanner** algorithm. As **PPlanner** **CPlanner** starts with the extraction of the strategy information and the initial task in step 1. In particular, step 1 extracts the action transfer procedures $\mathcal{TP}$ and sets the algorithm variable $\vec{A}_{add}$ to the empty list. In this variable **CPlanner** stores the actions, which it introduces successively. Afterwards, step 2 and step 3 check whether **CPlanner** terminates successfully or interrupts. These two steps equal step 2 and step 3 of **PPlanner**, respectively, given in Figure 27.

Step 4 first evaluates the control rules of kind 'TransferProcedure' on the alternative action transfer procedures $\mathcal{TP}$. This results in a changed and re-ordered alternative list $\mathcal{TP}_{rest}$. Then, step 4 evaluates the action transfer procedures in the order of this list until either one procedure provides an action or a demand, which is stored in the algorithm variable Obj, or all procedures have been tried. That is, at the end of step 4 Obj is either bound to an action A_{add} or to a demand D_{add} or it is unbound. These three cases are covered by the following steps, respectively. Step 5 describes the processing of an action A_{add}. In this case, **CPlanner** introduces A_{add} into the proof plan under construction employing the function Φ_{MULTI}. Moreover, it adds a history entry for the introduced action and passes new constraints to external constraint solvers. Additionally, the introduced action is added to $\vec{A}_{add}$. Then, **CPlanner** continues with step 2. Step 6 processes a demand D_{add}. It writes the demand onto the control blackboard and terminates then with an interrupt message. If the evaluation of the action transfer procedure provides neither an action nor a demand, then **CPlanner** terminates in step 7 with a failure message. This step equals step 7 of **PPlanner** in Figure 27.

5.5.4 The InstMeta Algorithm

Strategies of the algorithm **InstMeta** tackle an instantiation-task and compute a binding for the meta-variable of the instantiation-task. With this new binding a new binding store is created, which is added to the sequence of binding stores on the proof blackboard. A strategy of **InstMeta** specifies one parameter, namely a function that determines how the instantiation for a meta-variable is computed. We discussed some strategies of **InstMeta** in section 4.1.

Figure 29 contains a pseudo-code description of **InstMeta**. **InstMeta** has two arguments. First, a pair (S, T), which consists of an **InstMeta** strategy S and an instantiation-task T. Second, the list of all alternative job offers on the control blackboard, when the **InstMeta** strategy was invoked. **InstMeta** returns no specific output but updates the content of the proof blackboard.

Step 1 in Figure 29 applies the instantiation computation function of the strategy S to the task T. This function application can either succeed or fail. If the function application succeeds, then the algorithm variable $inst$ is bound to the returned value. Otherwise $inst$ stays unbound. Step 2 computes an instantiation action when $inst$ is bound and applies this action with Φ_{MULTI} to the strategic proof plan elements on the proof blackboard. Finally, step 3 adds a new strategy-application record to the history on the proof blackboard. The execution message of this record entry depends on whether $inst$ is bound or not. When $inst$ is bound **InstMeta** creates a success message, otherwise **InstMeta** creates a failure message.

Currently, the computation function of an **InstMeta** strategy is provides either one (success) or no (failure) solution. This was sufficient for the case studies conducted so far. When it turns out that a set of alternative instantiations and reasoning on the selection of one alternative is needed, then **InstMeta** can easily be extended to cover this functionality: The variable $inst$ has to store a list of alternatives. Moreover, between step 1 and step 2 an additional step is needed, which

Input: (1) a pair (S, T) where S is a **INSTMETA** strategy and T is an instantiation-task, (2) the list of all alternative job offers $\mathcal{J}$.

Output: No output, only changes of the blackboards.

```
Algorithm: INSTMETA$((S,T),\mathcal{J})$

  1. Compute Instantiation
     Let $inst$:=apply(compute-inst-function$(S), T)$.

  2. Compute and Apply Action
     If bound$(inst)$
        then
           Let $A_{add}$:=new-instmeta-action$(S, T, inst)$.
           PB:=$\Phi_{\text{MULTI}}(A_{add}, PB)$.

  3. Update History
     If bound$(inst)$
        than
           Let $message$:=create-success-message$(S, T)$.
           add-strategy-application-record-to-history$(\mathcal{J}, A_{add}, \emptyset, message)$.
        else
           Let $message$:=create-failure-message$(S, T)$.
           add-strategy-application-record-to-history$(\mathcal{J}, \emptyset, \emptyset, message)$.
     Terminate.
```

Figure 29: The **INSTMETA** algorithm.

evaluates control rules on the alternative instantiations and selects one. The control rules would become an additional parameter of **INSTMETA**.

5.5.5 The ATP Algorithm

Strategies of the algorithm **ATP** refine a strategic proof plan by solving a line-task with an **ATP** action. They apply external automated theorem provers and check whether their output is a proof. A strategy of **ATP** specifies two parameters for these two functionalities, namely an application function and an output check function. We discussed a strategy of **ATP** in section 4.4.

Figure 30 contains a pseudo-code description of the **ATP** algorithm. **ATP** has two arguments. First, a pair (S, T), which consists of an **ATP** strategy S and an instantiation-task T. Second, the list of all alternative job offers on the control blackboard, when the **ATP** strategy was invoked. **ATP** returns no specific output but updates the content of the proof blackboard.

Step 1 applies the application function of the strategy S to the task T. This function application provides an output, which is stored in the algorithm variable out. Step 2 applies the output check function to out, which returns either $true$ or nil. If the result, which is stored in the algorithm variable $check$, is $true$, then out is accepted as proof. In this case, **ATP** computes an action and applies this action with Φ_{MULTI} to the strategic proof plan elements on the proof blackboard

```
Input: (1) a pair $(S,T)$ where $S$ is an ATP and $T$ is a line-task, (2) the list of all alternative job offers $\mathcal{J}$.

Output: No output, only changes of the blackboards.

Algorithm: ATP$((S,T),\mathcal{J})$

  1. Apply Provers
     Let $out$:=apply(atp-apply-function$(S),T)$.

  2. Check Output
     Let $check$:=apply(atp-output-check-function$(S), out, T)$.

  3. Compute and Apply Action
     If $check = true$
       then
         Let $A_{add}$:=new-atp-action$(S,T,out)$.
         PB:=$\Phi_{\text{MULTI}}(A_{add}, PB)$.

  4. Update History
     If $check = true$
       then
         Let $message$:=create-success-message$(S,T)$.
         add-strategy-application-record-to-history$(\mathcal{J}, A_{add}, \emptyset, message)$.
       else
         Let $message$:=create-failure-message$(S,T)$.
         add-strategy-application-record-to-history$(\mathcal{J}, \emptyset, \emptyset, message)$.
     Terminate.
```

Figure 30: The **ATP** algorithm.

(see step 3 in Figure 30). Finally, step 4 adds a new strategy-application record to the history on the proof blackboard. The execution message of this record entry depends on whether $check$ is $true$. If $check$ is $true$, then **ATP** creates a success message, otherwise it creates a failure message.

5.5.6 The EXP Algorithm

The algorithm **EXP** refines a strategic proof plan by expanding complex steps. When applied to a closed proof line L whose justification is $(J\ P_1\ \ldots\ Pn)$, then **EXP** computes a proof segment that derives L from $P_1, \ldots, P_n$ at a lower level of abstraction. **EXP** has no parameters. The only strategy of **EXP** is ExpS.

Figure 31 contains a pseudo-code description of the **EXP** algorithm. **EXP** obtains two arguments. First, a pair (S,T), which consists of a **EXP** strategy S (i.e., ExpS) and an expansion-task T. Second, the list of all alternative job offers on the control blackboard, when the **EXP** strategy was invoked. **EXP** returns no specific output but updates the content of the proof blackboard.

Step 1 tests whether the justification **EXP** should expand is a tactic application or a method application. Depending on what kind of step it finds **EXP** employs either the function *expand-tactic* or the function *expand-method* to compute the expansion proof segment. *expand-tactic* evaluates

Input: (1) a pair (S,T) where S is an **EXP** strategy and $T = L|^{Exp}$ is an expansion-task, (2) the list of all alternative job offers $\mathcal{J}$.

Output: No output, only changes of the blackboards.

```
Algorithm: EXP$((S,T),\mathcal{J})$

  1. Compute Expansion-Segment
     Let $(J\ P_1\ \ldots\ P_n)$ be the justification of $L$.
     If is-tactic$(J)$
       then
         Let $exp\text{-}segment$:=expand-tactic$(L)$.
       else
         Let $exp\text{-}segment$:=expand-method$(L)$.

  2. Compute and Apply Action
     If bound$(exp\text{-}segment)$
       then
         Let $A_{add}$:=new-expansion-action$(S,T,exp\text{-}segment)$.
         PB:=$\Phi_{\text{MULTI}}(A_{add},PB)$.


  3. Update History
     If bound$exp\text{-}segment$
       then
         Let $message$:=create-success-message$(S,T)$.
         add-strategy-application-record-to-history$(\mathcal{J},A_{add},\emptyset,message)$.
       else
         Let $message$:=create-failure-message$(S,T)$.
         add-strategy-application-record-to-history$(\mathcal{J},\emptyset,\emptyset,message)$.
     Terminate.
```

Figure 31: The **EXP** algorithm.

the expansion procedure of the found tactic whereas *expand-method* instantiates the proof schema of the found method. When these function applications succeed, then the algorithm variable $exp\text{-}segment$ is bound to the computed proof segment. Otherwise $exp\text{-}segment$ stays unbound. When $exp\text{-}segment$ is bound, Step 2 creates an expansion action and applies the action with Φ_{MULTI} to the elements of the strategic proof plan on the proof blackboard. Afterwards, step 3 adds a new strategy-application record to the history on the proof blackboard. The execution message of this record entry depends on whether $exp\text{-}segment$ is bound or not. When $exp\text{-}segment$ is bound **EXP** creates a success message, otherwise **EXP** creates a failure message.

5.5.7 The BACKTRACK Algorithm

BACKTRACK is an algorithm that removes the actions introduced by other algorithms of MULTI from a strategic proof plan. **BACKTRACK** adds no own actions but only history entries. When to backtrack and which actions to backtrack is not hard-wired in the MULTI algorithm but is subject of the different strategies of **BACKTRACK** and the guidance by reasoning at the strategy-level. A strategy of **BACKTRACK** specifies a function that selects the set of actions in the current strategic proof plan that should be deleted. When MULTI invokes a **BACKTRACK** strategy, then **BACKTRACK** removes all actions explicitly selected by this function as well as all actions that depend from these actions. Thus, the backtracking in MULTI is dependency-directed in the sense discussed in section 3. We described a strategy of **BACKTRACK** in section 4.1.

Before we give a pseudo-code description of the **BACKTRACK** algorithm we shall introduce the notion of dependency among actions and when an action is deletable. Both notions are extensions of the concepts introduced for PLAN in section 3.3. When an action is introduced into a strategic proof plan, then it modifies the elements of the strategic proof plan. Other actions introduced later on may depend on these modifications. For instance, when a method action introduces a new proof line, which is used later on by another action, then the second action is not possible without the first action. In the following definition, we shall define for the different kinds of strategic actions and for method actions which other actions in an action sequence depend on them.

Definition Let $\vec{A}$ be a sequence of actions with

$$\vec{A}=[A_1,\ldots,A_{i-1},A_i,A_{i+1},\ldots,A_n].$$

The set of actions in $\vec{A}$, which *depend* on A_i is defined for the different kinds of actions in MULTI as follows.

Method Action: Let A_i be a method action with the $\ominus$ conclusions $\ominus Concs$, the $\oplus$ conclusions $\oplus Concs$, and the $\oplus$ premises $\oplus Prems$. If A_i contains some binding constraints, then $\{A_{i+1},\ldots,A_n\}$ depend on A_i. Otherwise, $A_j \in \{A_{i+1},\ldots,A_n\}$ depends on A_i if:

1. A_j is a method action whose sets of conclusions or premises contains a proof line of $\oplus Concs$ or $\oplus Prems$ (which are the new proof lines introduced by A_i),
2. A_j is an **INSTMETA** action, which tackles an instantiation-task whose meta-variable is introduced by A_i,
3. A_j is an **EXP** action, which tackles an expansion-task whose proof line is in $\ominus Concs$ or $\oplus Concs$ (the proof lines closed by A_i),
4. A_j is an **ATP** action, which tackles a line-task that contains either as support or as conclusion a proof line of $\oplus Concs$ or $\oplus Prems$,
5. A_j is a **PPLANNER** or **CPLANNER** action, which contains an action that depends on A_i.

INSTMETA Action: Let A_i be an **INSTMETA** action. Then $\{A_{i+1},\ldots,A_n\}$ depend on A_i.

ATP Action: Let A_i be an **ATP** action. $A_j \in \{A_{i+1},\ldots,A_n\}$ depends on A_i if A_j is an **EXP** action, which tackles the expansion-task with the proof line closed by A_i.

EXP Action: Let A_i be an **EXP** action with the set $\mathcal{L}_{new}$ of new proof lines in the proof-segment. Let $T = L|^{Exp}$ be the task of A_i. Then $A_j \in \{A_{i+1},\ldots,A_n\}$ depends on A_i if

1. A_j is a method action, which contains either as conclusion or as premise a proof line of $\mathcal{L}_{new}$, or which contains L as $\ominus$ conclusion,[28]

[28] If A_i opens L again, then L can be closed again later on by another method action.

2. A_j is an **InstMeta** action, which tackles an instantiation-task whose meta-variable is introduced by A_i,
3. A_j is an **Exp** action, which tackles an expansion-task whose proof line is in $\mathcal{L}_{new}$,
4. A_j is an **ATP** action, which tackles a line-task that contains a proof line of $\mathcal{L}_{new}$ either as support or as goal, or which tackles a line-task whose goal is L,[28]
5. A_j is a **PPlanner** or **CPlanner** action, which contains an actions that depends on A_i.

CPlanner or PPlanner Action: Let A_i be a **CPlanner** or a **PPlanner** action whose sequence of actions is $[A'_1, \ldots, A'_m]$. Then $A_j \in \{A_{i+1}, \ldots, A_n\}$ depends on A_i if there is an action $A'_k \in [A'_1, \ldots, A'_m]$ such that A_j depends on A'_k.

Finally, we have to define which actions of an action sequence depend on an action that is contained within a **CPlanner** or **PPlanner** action:
Let A_i be a **CPlanner** or **PPlanner** action with action sequence:

$$[A'_1, \ldots, A'_{i-1}, A'_i, A'_{i+1}, \ldots, A'_n].$$

Then the set of actions that depend on A'_i with respect to $\vec{A}$ is the set of actions that depend on A'_i with respect to the action sequence $[A_1, \ldots, A_{i-1}] \cup [A'_1, \ldots, A'_{i-1}, A'_i, A'_{i+1}, \ldots, A'_n] \cup [A_{i+1}, \ldots, A_n]$. □

Note that with this definition all actions succeeding an action that introduces new bindings (i.e., method actions with bindings and **InstMeta** actions) depend on this action. We use now the notion of dependency of actions to define when an action is deletable with respect to an action sequence.

Definition Let $\vec{A}$ be a sequence of actions with

$$\vec{A}=[A_1, \ldots, A_{i-1}, A_{del}, A_{i+1}, \ldots, A_n].$$

A_{del} is *deletable* with respect to $\vec{A}$ if the set of actions in $\vec{A}$ that depend on A_{del} is empty. □

Next, we define the functions Φ^{-1}_{MULTI} and $\vec{\Phi}^{-1}_{\text{MULTI}}$, which delete actions.[29] We give first the general outline of Φ^{-1}_{MULTI} and define the recursive $\vec{\Phi}^{-1}_{\text{MULTI}}$. Afterwards, we define Φ^{-1}_{MULTI} for the different kinds of actions.

Definition The *action deletion function* Φ^{-1}_{MULTI} is a partial function that maps a sequence of actions, an agenda, a $\mathcal{PDS}$, a sequence of binding stores and an action into a sequence of actions, an agenda, a $\mathcal{PDS}$, and a sequence of binding stores, i.e.,

$$\Phi^{-1}_{\text{MULTI}} : \vec{A} \times \hat{\text{A}} \times \mathcal{P} \times \vec{\mathcal{B}}\text{S} \times A_{del} \mapsto \vec{A}\text{'} \times \hat{\text{A}}\text{'} \times \mathcal{P}\text{'} \times \vec{\mathcal{B}}\text{S'}.$$

The *recursive action deletion function* $\vec{\Phi}^{-1}_{\text{MULTI}}$ is a partial function that maps a sequence of actions, an agenda, a $\mathcal{PDS}$, a sequence of binding stores, and a sequence of actions into a sequence of actions, an agenda, a $\mathcal{PDS}$, and a sequence of binding stores, i.e.,

$$\vec{\Phi}^{-1}_{\text{MULTI}} : \vec{A} \times \hat{\text{A}} \times \mathcal{P} \times \vec{\mathcal{B}}\text{S} \times \vec{A}_{del} \mapsto \vec{A}\text{'} \times \hat{\text{A}}\text{'} \times \mathcal{P}\text{'} \times \vec{\mathcal{B}}\text{S'}.$$

[29] Since action deletion is conceptually the inverse operation of action introduction we call these functions Φ^{-1}_{MULTI} and $\vec{\Phi}^{-1}_{\text{MULTI}}$ although technically they are not the inverse functions of Φ_{MULTI} and $\vec{\Phi}_{\text{MULTI}}$.

$\vec{\Phi}^{-1}_{\text{MULTI}}$ is recursively defined as follows.
Let $\vec{A}$ be a sequence of actions, Â an agenda, $\mathcal{P}$ a $\mathcal{PDS}$, $\vec{\mathcal{B}}$S a sequence of binding stores, and $\vec{A}_{del}$ a sequence of actions.

1. If $\vec{A}_{del}$ is empty, then
   $\vec{\Phi}^{-1}_{\text{MULTI}}(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, \vec{A}_{del}) := (\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S})$.
2. Otherwise let $A_{del} := \textit{first}(\vec{A}_{del})$ and $\vec{A}\text{'}_{del} := \textit{rest}(\vec{A}_{del})$. If A_{del} is in $\vec{A}$ or part of a **CPLANNER** or **PPLANNER** action in $\vec{A}$ and A_{del} is deletable with respect to $\vec{A}$, then
   $\vec{\Phi}^{-1}_{\text{MULTI}}(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, \vec{A}_{del}) := \vec{\Phi}^{-1}_{\text{MULTI}}(\Phi^{-1}_{\text{MULTI}}(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, A_{del}), \vec{A}\text{'}_{del})$.

□

In the single definitions of the function Φ^{-1}_{MULTI} for the different kinds of actions we describe the modifications of the sequence of actions, the agenda, the $\mathcal{PDS}$, and the sequence of binding stores caused by the deletion of a respective action. Although the notion of deletability of an action is only defined with respect to a sequence of actions, we assume that the agenda, the $\mathcal{PDS}$, and the sequence of binding stores are not arbitrary, but created by this sequence of actions (in particular, by the action that should be deleted).

We start with the definition of Φ^{-1}_{MULTI} for method actions. Since in MULTI the action sequences consist only of strategic actions, a method action can occur only within a **PPLANNER** or **CPLANNER** action. Hence, the following definition describes the deletion of a method action within a **PPLANNER** or **CPLANNER** action.

Definition Let $\vec{A}$ be a sequence of actions and let A_{del} be a method action, which is in an **PPLANNER** or **CPLANNER** action $A_{planner}$ in $\vec{A}$, i.e., $\vec{A}$=$[A_1, \ldots, A_{i-1}, A_{planner}, A_{i+1}, \ldots, A_n]$. Let $\vec{\mathcal{B}}$S be a sequence of bindings stores, $\mathcal{P}$ a $\mathcal{PDS}$, and Â an agenda. Moreover, let $\oplus Concs$ be the $\oplus$ conclusions, $\ominus Concs$ the $\ominus$ conclusions, $\oplus Prems$ the $\oplus$ premises, $\ominus Prems$ the $\ominus$ premises, and $BPrems$ the blank premises of A_{del}. Let $T = L \blacktriangleleft \mathcal{SUPPS}_L$ be the task of A_{del} and let σ be the binding constraints of A_{del}.
$Prems$:=$\oplus Prems \cup \ominus Prems \cup BPrems$,
$Concs$:=$\oplus Concs \cup \ominus Concs$
$Lines\text{-}To\text{-}Remove$:=$\oplus Concs \cup \oplus Prems$
$Old\text{-}Line\text{-}Tasks$:=$[L' \blacktriangleleft \mathcal{SUPPS}_{L'} \mid L' \in \oplus Prems]$.
$Old\text{-}Inst\text{-}Tasks$:=$[mv|^{Inst} \mid mv \in New\text{-}Lines\ and\ nowhere\ else\ in\ \mathcal{P}]$.
$Old\text{-}Exp\text{-}Tasks$:=$[C|^{Exp} \mid C\ in\ Concs]$.
$Tasks\text{-}To\text{-}Remove$:= $Old\text{-}Line\text{-}Tasks \cup Old\text{-}Inst\text{-}Tasks \cup Old\text{-}Exp\text{-}Tasks.$
$New\text{-}Inst\text{-}Tasks$:=$[mv|^{Inst} \mid mv\ bound\ in\sigma]$.
$New\text{-}Tasks$:=$[T] \cup New\text{-}Inst\text{-}Tasks.$

If A_{del} is deletable with respect to $\vec{A}$ and if Â, $\mathcal{P}$, and $\vec{\mathcal{B}}$S resulted from the introduction of $\vec{A}$ (to some agenda, $\mathcal{PDS}$, and sequence of binding stores), then the result $(\vec{A}\text{'}, \hat{\text{A}}\text{'}, \mathcal{P}\text{'}, \vec{\mathcal{B}}\text{S'})$ of $\Phi^{-1}_{\text{MULTI}}(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, A_{del})$ is defined by:

- $\vec{A}$':= $[A_1, \ldots, A_{i-1}, A'_{planner}, A_{i+1}, \ldots, A_n]$
  where $A'_{planner}$ results from $A_{planner}$ by removing A_{del} from the sequence of actions of $A_{planner}$.

- $\hat{A}$':= $New\text{-}Tasks \cup (\hat{A} - Tasks\text{-}To\text{-}Remove)$.
- $\mathcal{P}$' results from $\mathcal{P}$ by
  1. removing the lines $Lines\text{-}To\text{-}Remove$ and
  2. justifying the proof lines $\ominus Concs$ with $Open$, respectively.
- If σ is empty, then $\vec{\mathcal{B}}$S':=$\vec{\mathcal{B}}$S, otherwise $\vec{\mathcal{B}}$S':=$\vec{\mathcal{B}}\text{S} - \mathit{last}(\vec{\mathcal{B}}\text{S})$.[30]

□

Definition Let $\vec{A}$ be a sequence of actions and let A_{del} be an **INSTMETA** action in $\vec{A}$. Let $\vec{\mathcal{B}}$S be a sequence of bindings stores, $\mathcal{P}$ a $\mathcal{PDS}$, and $\hat{A}$ an agenda.

If A_{del} is deletable with respect to $\vec{A}$ and if $\hat{A}$, $\mathcal{P}$, and $\vec{\mathcal{B}}$S resulted from the introduction of $\vec{A}$ (to some agenda, $\mathcal{PDS}$, and sequence of binding stores), then the result $(\vec{A}', \hat{A}', \mathcal{P}', \vec{\mathcal{B}}\text{S}')$ of $\Phi^{-1}_{\text{MULTI}}(\vec{A}, \hat{A}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, A_{del})$ is defined by:

- $\vec{A}$':=$\vec{A} - A_{del}$.
- $\hat{A}$':=$\hat{A} \cup [T]$ where T is the task of A_{del}.
- $\mathcal{P}$':=$\mathcal{P}$.
- $\vec{\mathcal{B}}$S':=$\vec{\mathcal{B}}\text{S} - \mathit{last}(\vec{\mathcal{B}}\text{S})$.

□

[30] If σ is not empty, then the last binding store in $\vec{\mathcal{B}}$S has to be the binding store resulting from the introduction of A_{del} since otherwise A_{del} would not be deletable. Thus, when A_{del} is deleted, then the last binding store has to be removed.

Definition Let $\vec{A}$ be a sequence of actions and let A_{del} be an **ATP** action in $\vec{A}$. Let $\vec{\mathcal{B}}$S be a sequence of bindings stores, $\mathcal{P}$ a $\mathcal{PDS}$, and Â an agenda. Let $T = L \blacktriangleleft \mathcal{SUPPS}_L$ be the task of A_{del}.

If A_{del} is deletable with respect to $\vec{A}$ and if Â, $\mathcal{P}$, and $\vec{\mathcal{B}}$S resulted from the introduction of $\vec{A}$ (to some agenda, $\mathcal{PDS}$, and sequence of binding stores), then the result $(\vec{A}\text{'}, \text{Â'}, \mathcal{P}\text{'}, \vec{\mathcal{B}}\text{S'})$ of $\Phi^{-1}_{\mathrm{MULTI}}(\vec{A}, \text{Â}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, A_{del})$ is defined by:

- $\vec{A}$':=$\vec{A} - A_{del}$.
- Â':=(Â $\cup$ $[T]$) $-$ $L|^{Exp}$.
- $\mathcal{P}$' results from $\mathcal{P}$ by opening the line L.
- $\vec{\mathcal{B}}$S':=$\vec{\mathcal{B}}$S.

□

Definition Let $\vec{A}$ be a sequence of actions and let A_{del} be an **EXP** action in $\vec{A}$. Let $\vec{\mathcal{B}}$S be a sequence of bindings stores, $\mathcal{P}$ a $\mathcal{PDS}$, and Â an agenda. Moreover, let $T = L|^{Exp}$ be the task of A_{del} and $(J\ P_1\ \ldots Pn)$ the justification of L at the next higher level of abstraction (i.e., the justification of L before A_{del} was performed).
$Lines\text{-}To\text{-}Remove$:=$\{L'|L' \in expansion\text{-}segment\ of\ A_{del}\} - \{L, P_1, \ldots, P_n\}$.
$New\text{-}Tasks$:=$[T]$.
$Old\text{-}Open\text{-}Lines$:=$\{L'|L' \in open\text{-}lines\ of A_{add}\}$.
$Old\text{-}Line\text{-}Tasks$:=$[L' \blacktriangleleft \mathcal{SUPPS}_{L'} \mid L'\ in\ Old\text{-}Open\text{-}Lines]$.
$Old\text{-}Inst\text{-}Tasks$:=$[mv|^{Inst} \mid mv \in\ Lines\text{-}To\text{-}Remove\ and\ nowhere\ else\ in\ \mathcal{PDS}]$.
$Old\text{-}Exp\text{-}Tasks$:=
$[L'|^{Exp}\ |(L' \in\ Lines\text{-}To\text{-}Remove\ or\ L' = L)\ and\ L'\ closed\ by\ tactic]$.
$Tasks\text{-}To\text{-}Remove$:= $Old\text{-}Line\text{-}Tasks \cup Old\text{-}Inst\text{-}Tasks \cup Old\text{-}Exp\text{-}Tasks$.

If A_{del} is deletable with respect to $\vec{A}$ and if Â, $\mathcal{P}$, and $\vec{\mathcal{B}}$S resulted from the introduction of $\vec{A}$ (to some agenda, $\mathcal{PDS}$, and sequence of binding stores), then the result $(\vec{A}\text{'}, \text{Â'}, \mathcal{P}\text{'}, \vec{\mathcal{B}}\text{S'})$ of $\Phi^{-1}_{\mathrm{MULTI}}(\vec{A}, \text{Â}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, A_{del})$ is defined by:

- $\vec{A}$':=$\vec{A} - A_{del}$.
- Â':= $New\text{-}Tasks$ $\cup$ (Â $-$ $Tasks\text{-}To\text{-}Remove$).
- $\mathcal{P}$' results from $\mathcal{P}$ by
  1. removing the current justification from L and setting $(J\ P_1\ \ldots Pn)$ as the current one, and
  2. removing the proof lines in $New\text{-}Lines$.
- $\vec{\mathcal{B}}$S':=$\vec{\mathcal{B}}$S.

□

Definition Let $\vec{A}$ be a sequence of actions and let A_{del} be a **CPLANNER** or a **PPLANNER** action in $\vec{A}$. Let $\vec{\mathcal{B}S}$ be a sequence of bindings stores, $\mathcal{P}$ a $\mathcal{PDS}$, and Â an agenda. Moreover, let $[A_1, \ldots, A_n]$ be the action-sequence of A_{del}.
$(\vec{A}_{rec}, \hat{\mathrm{A}}_{rec}, \mathcal{P}_{rec}, \vec{\mathcal{B}S}_{rec}) := \vec{\Phi}^{-1}_{\mathrm{MULTI}}(\vec{A}, \hat{\mathrm{A}}, \mathcal{P}, \vec{\mathcal{B}S}, [A_n, \ldots, A_1])$
If A_{del} is deletable with respect to $\vec{A}$ and if Â, $\mathcal{P}$, and $\vec{\mathcal{B}S}$ resulted from the introduction of $\vec{A}$ (to some agenda, $\mathcal{PDS}$, and sequence of binding stores), then the result $(\vec{A}', \hat{\mathrm{A}}', \mathcal{P}', \vec{\mathcal{B}S}')$ of $\Phi^{-1}_{\mathrm{MULTI}}(\vec{A}, \hat{\mathrm{A}}, \mathcal{P}, \vec{\mathcal{B}S}, A_{del})$ is defined by:

- $\vec{A}' := \vec{A}_{rec} - [A_{del}]$.[31]
- $\hat{\mathrm{A}}' := \hat{\mathrm{A}}_{rec}$.
- $\mathcal{P}' := \mathcal{P}_{rec}$.
- $\vec{\mathcal{B}S}' := \vec{\mathcal{B}S}_{rec}$.

□

With these definitions at our disposal, we can now describe the **BACKTRACK** algorithm. Figure 32 contains a pseudo-code description of **BACKTRACK**. **BACKTRACK** obtains two arguments. First, a pair (S, T), which consists of a **BACKTRACK** strategy S and a task T. Second, the list of all alternative job offers on the control blackboard, when the **BACKTRACK** strategy was invoked. **BACKTRACK** returns no specific output but updates the content of the proof blackboard.

Step 1 applies the computation function of the strategy S to the task T. This returns a sequence of actions that **BACKTRACK** should delete, and **BACKTRACK** binds the algorithm variable $\vec{A}_{del}$ to this action sequence. Moreover, **BACKTRACK** writes a backtrack-start entry with this information to the history.

The steps 2-5 are essentially a while-loop, which is passed through until $\vec{A}_{del}$ is empty. First, Step 2 checks whether $\vec{A}_{del}$ is empty. If this is the case, it creates a success message,[32] writes a backtrack-stop entry with this message to the history, and terminates. Otherwise, step 3 picks the first action from $\vec{A}_{del}$ and stores it in the algorithm variable A_{del}. A_{del} is then either deleted in step 5 or step 4 extends $\vec{A}_{del}$ depending on A_{del}. Step 4 first checks whether A_{del} is deletable with respect to the sequence of actions on the proof blackboard. If this is not the case, then there are actions which depend on A_{del} and step 4 adds these actions, which are computed by the function *dependend-actions*, in front of $\vec{A}_{del}$. If A_{del} is deletable, then step 4 checks next whether it is an action of **PPLANNER** or **CPLANNER** whose action-sequence is not empty. If this holds, then it adds the action sequence of A_{del} in front of $\vec{A}_{del}$. Otherwise, step 5 is reached, which uses $\Phi^{-1}_{\mathrm{MULTI}}$ to delete A_{del} and to update the action sequence, the agenda, the $\mathcal{PDS}$, and the sequence of binding stores on the proof blackboard. Moreover, it adds an action-deletion entry to the history and removes A_{del} from $\vec{A}_{del}$.

If the deleted action A_{del} belongs to a terminated **PPLANNER** or **CPLANNER** strategy execution (this is checked by the function *action-of-terminated-strategy*), then a re-invocation of this strategy execution should be enabled again. **BACKTRACK** re-activates the strategy execution by writing an

[31] When all actions in A_{del} are deleted, then A_{del} remains with an empty action sequence. Here A_{del} itself is deleted from the action sequence.

[32] Note that **BACKTRACK** is not supposed to fail (except of hopefully not occurring programming errors).

Input: (1) a pair (S, T) where S is a BACKTRACK strategy and T is a task, (2) the list of all alternative job offers $\mathcal{J}$.

Output: No output, only changes of the blackboards.

```
Algorithm: BACKTRACK$((S, T), \mathcal{J})$

  1. Compute Actions To Be Deleted
     Let $\vec{A}_{del}$:=apply(compute-del-actions-function$(S), T)$.
     add-backtrack-start-record-to-history$(\mathcal{J}, \vec{A}_{del})$.

  2. Terminate
     If $\vec{A}_{del}=\emptyset$
        then
           Let $message$:=create-success-message$(S, T)$.
           add-backtrack-stop-record-to-history$(message)$.
           Terminate.

  3. Select Action
     Let $A_{del}$:=first$(\vec{A}_{del})$.

  4. Extend Actions
     If $A_{del}$ is not deletable wrt. the sequence of actions on PB
        then
           $\vec{A}_{del}$:=dependend-actions$(A_{del}) \cup \vec{A}_{del}$.
           Goto step 3.

     If $A_{del}$ is CPLANNER or PPLANNER action, whose action-sequence is not empty
        then
           $\vec{A}_{del}$:=action-sequence$(A_{del}) \cup \vec{A}_{del}$.
           Goto step 3.

  5. Delete Action
     PB:=$\Phi^{-1}_{\text{MULTI}}(A_{del}, PB)$.
     add-action-del-record($A_{del}$).
     Let $\vec{A}_{del}$:=$\vec{A}_{del} - [A_{del}]$.
     If action-of-terminated-strategy$(A_{del})$
        then
           write-to-memory(get-tasktag$(A_{del}), \emptyset)$.
     Goto step 2.
```

Figure 32: The BACKTRACK algorithm.

entry to the memory consisting of the task tag of the strategy execution (which is computed by the function *get-tasktag* from the history) and an empty set of demand pointers. From this memory entry the terminated strategy execution can be re-invoked.

Note that BACKTRACK could apply Φ^{-1}_{MULTI} directly to actions of PPLANNER and CPLANNER that are not empty (since we did define Φ^{-1}_{MULTI} for such actions in definition 5.5.7). However, **BackTrack** first successively deletes the action sequence of an action of **PPlanner** and **CPlanner** before it deletes the "empty" **PPlanner** or **CPlanner** action. This guarantees that detailed history information for each deleted action is created (i.e, for each action, which is in the action-sequence of an action of **PPlanner** or **CPlanner** as well as for the **PPlanner** or **CPlanner** action itself).

5.6 Remarks

5.6.1 Representing the Search with Trees

The check for dependency among actions as well as the changes caused by backtracking of an action are complex operations as described in the previous section. The problem is that the $\mathcal{PDS}$, which is the central data structure in the current implementation of ΩMEGA and MULTI, is a complex data structure difficult to maintain. In the ongoing re-implementation of the ΩMEGA system on top of the CORE system [4] we suggest an agenda as the (only) central data structure. Moreover, we suggest additional data structures to considerably simplify the backtracking of actions.

The introduction of an action into a strategic proof plan reduces a task to a set of tasks, which can be empty. The introduced actions and the resulting tasks could be stored in a tree, a so-called *task-action-tree*, whose nodes are labeled with the tasks and whose edges are labeled with the actions.[33] Figure 33 depicts such a task-action-tree. The root node of the tree is labeled with the initial task. If this tree is constructed during the strategic proof planning process, then the current agenda consists always of the tasks of the leave nodes of the tree.

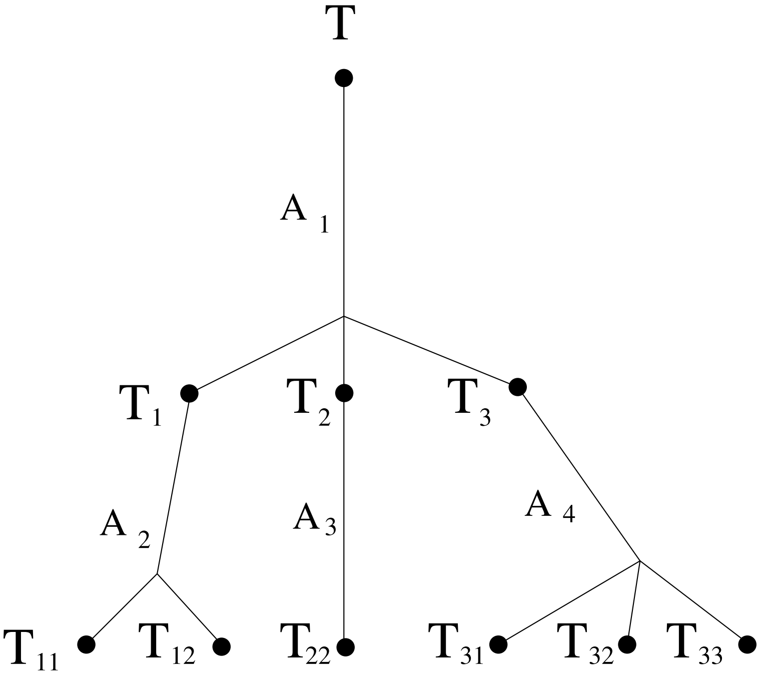


Figure 33: A task-action-tree.

[33] Actually, we use multi-edges that connect one parent node with several children nodes.

With a task-action-tree the dependency among actions could be formulated as follows: An action A_i depends on another action A_j if the path from the root node to A_i contains A_j. The changes caused by the backtracking of an action could also be stated simpler than currently: If a deletable action A is backtracked, then the children tasks of the action A are removed and the parent task is introduced again into the agenda.

5.6.2 Creating Different Kinds of Solution Proof Plans

In section 5.3, we defined three different notions of strategic solution proof plans, namely method-level solution proof plans, instantiated method-level solution proof plans, and full solution proof plans. In order to produce a method-level solution proof plan Multi can ignore the instantiation tasks and the expansion-tasks; to produce an instantiated method-level solution proof plans Multi can ignore only the expansion-tasks; to create a full solution proof plan Multi has to tackle all kinds of tasks.

The simplest possibility to make Multi search for a particular kind of solution proof plan is to prohibit some strategies. For instance, if there are no strategies of **Exp**, then expansion-tasks will be ignored and Multi will search for an instantiated method-level solution proof plan. In the case studies it turned out that this approach has the drawback that expansion-tasks are created although they are ignored later on. Therefore, we avoid the creation of not desired expansion-tasks. The user can declare methods or tactics whose applications he wants to be expanded by Multi as *not-reliable*. Multi creates expansion-tasks only for such proof lines L whose justification $(J\ P_1 \ldots P_N)$ uses a not-reliable method or tactic J.

5.6.3 Cooperation with Constraint Solvers

So far, the only constraint solver connected with Multi is $\mathcal{C}o\mathcal{SIE}$. Multi communicates directly with $\mathcal{C}o\mathcal{SIE}$ by interfaces in methods and strategies. When a method action is introduced that contains constraints for $\mathcal{C}o\mathcal{SIE}$, then these constraints are passed to $\mathcal{C}o\mathcal{SIE}$. Moreover, the two strategies InstIfDetermined and ComputeInstFromCS employ $\mathcal{C}o\mathcal{SIE}$ to obtain new bindings. If several constraint solvers should be connected with Multi, then a direct communication is not sufficient anymore. First, constraints should be passed to all connected constraint solvers for which they are relevant. Second, several constraint solvers should be able to directly exchange results without involving Multi.

As possible solution we suggest a *constraint solver coordination module*, which handles all communication and which stores all constraints and results. Each constraint solver that should be connected has to register by the coordination module. Multi passes new constraints to this module. Then, the module asks the connected constraints solvers whether this constraint is relevant for them and passes it to the relevant constraint solvers. The module performs the same distribution, if a constraint solver produces an intermediate result (i.e., when $\mathcal{C}o\mathcal{SIE}$ detects that the instantiation of meta-variable mv is already determined by its current constraints). When Multi backtracks and deletes some method actions with constraints, then the coordination module has also to organize the deletion of the constraints in the affected constraint solvers and the deletion of intermediate that depend on these constraints.

The module handles and distributes also queries of Multi. Multi passes queries (e.g., is the instantiation of meta-variable mv already determined?) only to the coordination module. Either the coordination module can answer the query directly (e.g., if an result passed by a connected constraint solver was already a unique instantiation for mv) or it distributes the query to the connected constraint solvers and passes the answer back to MULTI.

5.6.4 Dependencies in Backtracking

When the **BACKTRACK** algorithm removes an action, then it also removes all actions that depend on this action (see section 5.5.7). The notion of dependency for actions used by **BACKTRACK** (see definition 5.5.7) is strict and therefore **BACKTRACK** may removes more actions than necessary. In particular, the deletion of an **INSTMETA** action causes the deletion of all actions following this action in the current action sequence. We decided for this approach since a more detailed analysis of which following actions actually depend on a new binding is difficult and is still open.

Nevertheless, there are also dependencies between actions that are not covered by the dependency notion in definition 5.5.7. In particular, there can be various dependencies between actions that involve cooperation with constraint solvers (e.g., $\mathcal{C}o\mathcal{SIE}$). For instance, if the current constraints (e.g., $mv \leq t$ and $mv \geq t$) in $\mathcal{C}o\mathcal{SIE}$ determine the instantiation t for a meta-variable mv, then the strategy InstIfDetermined is applicable with respect to mv and introduces the binding $mv{:=}^b t$ into the strategic proof plan. Other actions can rely on this binding. When a method action that contains constraints for $\mathcal{C}o\mathcal{SIE}$ is backtracked, then mv may is not longer determined with respect to the resulting constraint store (e.g., if the constraint $mv \leq t$ is removed). In this case, the action of InstIfDetermined, which binds mv to t, has to be removed. Since this is not a problem of strategies of **INSTMETA** in general but of ComputeInstFromCS in particular, we did not implement such a dependency analysis into the **BACKTRACK** algorithm (i.e., it is not contained in the dependency notion introduced in definition 5.5.7). Rather we suggest to check such particular dependencies in strategic control rules that cause further backtracking.

The described situation is handled by the strategic control rule `check-det-insts`. `check-det-insts` checks whether the last strategy execution was a **BACKTRACK** step and whether it removed some method actions with constraints for $\mathcal{C}o\mathcal{SIE}$. If this is the case, it checks whether all actions of InstIfDetermined in the current sequence of actions are still valid in the sense that the meta-variables that they bind are still determined in $\mathcal{C}o\mathcal{SIE}$. Then, `check-det-insts` prefers backtracking for each action of InstIfDetermined that is no longer valid.

5.6.5 Failure Information in Execution Messages

When a strategy execution fails, then its algorithm creates a failure message. If possible the algorithm can attach information to a failure message, which can also be used by the control rules. For instance, **PPLANNER** can create and attach information why no applicable action could be found. This functionality affects many single steps in **PPLANNER** and in the procedures **CHOOSEACTION** and **CHOOSEACTIONALL**, which compute and select the next action to be applied. Hence, for the sake of simplicity and clarity, we did not describe this functionality in the algorithms themselves but give an informal description here.

That the procedures **CHOOSEACTION** and **CHOOSEACTIONALL** fail to to provide an action for a line-task T and a method M can be caused by three reasons:

Failed matching of proof lines The $\ominus$ conclusions of M do not match with the task line of T or the blank and $\ominus$ premises of M do not match with the supports of T.

Failed application conditions The evaluation of the application conditions of M can fail with respect to the substitution resulting from a successful matching of the proof lines of M with the task line and the supports of T.

Rejected actions Actions can be rejected by control rules or because they were already applied and then backtracked later on.

These tests are performed successively in **CHOOSEACTION** and **CHOOSEACTIONALL** in this order. Each time such a test fails, the function that performs the test creates an information record. For instance, when the function *eval-appl-conds* finds that the application condition App_c of method M fails with respect to the incomplete action A (which resulted from the successful matching of the proof lines of M with the proof lines of the given task), then *eval-appl-conds* creates the information record $applcondfailure(App_c, M, A)$. **CHOOSEACTION** and **CHOOSEACTIONALL** collect these information records and return them to **PPLANNER**. If there is no applicable action, then **PPLANNER** attaches the set of information records to the created failure message. Sample applications of MULTI that make use of such failure information are given in [35].

6 Conclusion

We presented the technical concepts underlying proof planning in the ΩMEGA system and gave detailed descriptions of the two proof planners in ΩMEGA, the simple proof planner PLAN and the multi-strategy proof planner MULTI. As sample application we discussed the application of both planners to the LIM+ example from the limit domain.

Since the LIM+ example can be solved by both planners it does not point out the advantages of proof planning with multiple-strategies as opposed to simple proof planning. We refer the interested reader to [34, 35] for a discussion of problems of the limit domain that cannot be solved by PLAN but by MULTI. The reason is that MULTI enables the flexible combination of different proof plan refinements (in particular flexible backtracking and meta-variable instantiation) guided by meta-reasoning. For further discussion of the application of PLAN to the limit domain see in the master thesis of Jürgen Zimmer [65] as well as in [42, 41, 45].

Further case studies performed with MULTI include applications to residue class problems [39, 36, 37, 38], to permutation group problems [13], and to the $\sqrt{2}$-is-irrational problem and similar problem [57]. A recent application of MULTI is its incorporation into the tutor system ACTIVEMATH [43] where it serves as the module to teach mathematical theorem proving [40]. In this application MULTI is not used as automated component (as in the other listed applications) but as mixed-initiative component that communicates with the user and incorporates the user to take decisions, i.e., which proof plan refinement to perform next.

A ChooseActionAll Algorithm

Input: (1) a task T, (2) a history $\vec{H}$, (3) a list of methods $\mathcal{M}$, (4) a list of control rules $\mathcal{C}$.

Output: Either a pair of an action and a list of actions or **fail**.

Algorithm: ChooseActionAll(T,$\vec{H}$,M,C)

Let T=$L_{open} \blacktriangleleft \mathcal{SUPPS}_{L_{open}}$.

```
1. Order Methods
   $Methods$:= evalcrules-methods($\mathcal{M}$,$\mathcal{C}$,$T$).
   Let $Methods$ = $[M_1, \ldots, M_n]$.
   When $Methods$ empty then terminate and return fail.
   $Actions_1$:=initial-action-set$(T, M_1)$.
               $\vdots$
   $Actions_n$:=initial-action-set$(T, M_n)$.

2. Handle Task, Supports, Parameters, and Appl. Conditions
   For $i = 1$ to $n$:

   (a) Match Task Line
       Let $\ominus Concs_i$ the $\ominus$ conclusions of $M_i$.
       $Actions_i$:=match-task-line($L_{open}$,$\ominus Concs_i$,$Actions_i$).
   (b) Select and Match Supports and Parameters
       Let $\ominus Prems_i$ and $BPrems_i$ the $\ominus$ premises and blank premises of $M_i$.
       Let $Params_i$ the parameter variables of $M_i$.
       $Supps$+$Params_i$:=evalcrules-s+p($\mathcal{SUPPS}_{L_{open}}$,$\mathcal{C}$,$T$,$M_i$,$Actions_i$).
       $Actions_i$:= match-s+p($Supps$+$Params_i$,$\ominus Prems_i \cup BPrems_i$,
                                                   $Params_i$,$Actions_i$).
   (c) Evaluate Application Conditions
       $Actions_i$:=eval-appl-conds($Actions_i$,$M_i$).

   $Actions$:=$Actions_1 \cup \ldots \cup Actions_n$.
   When $Actions$ empty then terminate and return fail.

3. Outline Computations
   eval-outline-computations($Actions$).
   complete-outline($Actions$).

4. Choose Action
   $Actions$:=remove-backtracked($Actions$,$\vec{H}$).
   $Actions$:=evalcrules-actions($Actions$,$\mathcal{C}$).
   If $Actions = \emptyset$
      then Terminate and return fail.
      else Terminate and return first($Actions$).
```

Figure 34: The **ChooseActionAll** algorithm.

B Lim+ Example

Lim_f.	Lim_f	$\vdash \lim_{x \to a} f(x) = l_f$	(Hyp)
Lim_g.	Lim_g	$\vdash \lim_{x \to a} g(x) = l_g$	(Hyp)
L_2.	Lim_f	$\vdash \forall \epsilon_1 \bullet (0 < \epsilon_1 \Rightarrow \exists \delta_1 \bullet (0 < \delta_1 \wedge \forall x_1 \bullet (\lvert x_1 - a \rvert < \delta_1 \wedge \lvert x_1 - a \rvert > 0 \Rightarrow \lvert f(x_1) - l_f \rvert < \epsilon_1)))$	(DefnUnfold-F Lim_f)
L_3.	Lim_g	$\vdash \forall \epsilon_2 \bullet (0 < \epsilon_2 \Rightarrow \exists \delta_2 \bullet (0 < \delta_2 \wedge \forall x_2 \bullet (\lvert x_2 - a \rvert < \delta_2 \wedge \lvert x_2 - a \rvert > 0 \Rightarrow \lvert g(x_2) - l_g \rvert < \epsilon_2)))$	(DefnUnfold-F Lim_g)
L_{17}.	Lim_f	$\vdash 0 < mv_{\epsilon_1} \Rightarrow \exists \delta_1 \bullet (0 < \delta_1 \wedge \forall x_1 \bullet (\lvert x_1 - a \rvert < \delta_1 \wedge \lvert x_1 - a \rvert > 0 \Rightarrow \lvert f(x_1) - l_f \rvert < mv_{\epsilon_1}))$	($\forall$E-F L_2)
L_{18}.	$\mathcal{H}_3$	$\vdash 0 < mv_{\epsilon_1}$	(TellCS-B)
L_{20}.	$\mathcal{H}_3$	$\vdash \exists \delta_1 \bullet (0 < \delta_1 \wedge \forall x_1 \bullet (\lvert x_1 - a \rvert < \delta_1 \wedge \lvert x_1 - a \rvert > 0 \Rightarrow \lvert f(x_1) - l_f \rvert < mv_{\epsilon_1}))$	($\Rightarrow_E$ L_{18} L_{17})
L_{21}.	L_{21}	$\vdash 0 < c_{\delta_1} \wedge \forall x_1 \bullet (\lvert x_1 - a \rvert < c_{\delta_1} \wedge \lvert x_1 - a \rvert > 0 \Rightarrow \lvert f(x_1) - l_f \rvert < mv_{\epsilon_1})$	(Hyp)
L_{23}.	L_{21}	$\vdash 0 < c_{\delta_1}$	($\wedge$E-F L_{21})
L_{24}.	L_{21}	$\vdash \forall x_1 \bullet (\lvert x_1 - a \rvert < c_{\delta_1} \wedge \lvert x_1 - a \rvert > 0 \Rightarrow \lvert f(x_1) - l_f \rvert < mv_{\epsilon_1})$	($\wedge$E-F L_{21})
L_{25}.	L_{21}	$\vdash \lvert mv_{x_1} - a \rvert < c_{\delta_1} \wedge \lvert mv_{x_1} - a \rvert > 0 \Rightarrow \lvert f(mv_{x_1}) - l_f \rvert < mv_{\epsilon_1}))$	($\forall$E-F L_{24})
L_{38}.	Lim_g	$\vdash 0 < mv_{\epsilon_2} \Rightarrow \exists \delta_2 \bullet (0 < \delta_2 \wedge \forall x_2 \bullet (\lvert x_2 - a \rvert < \delta_2 \wedge \lvert x_2 - a \rvert > 0 \Rightarrow \lvert g(x_2) - l_g \rvert < mv_{\epsilon_2}))$	($\forall$E-F L_3)
L_{39}.	$\mathcal{H}_3$	$\vdash 0 < mv_{\epsilon_2}$	(TellCS-B)
L_{41}.	$\mathcal{H}_3$	$\vdash \exists \delta_2 \bullet (0 < \delta_2 \wedge \forall x_2 \bullet (\lvert x_2 - a \rvert < \delta_2 \wedge \lvert x_2 - a \rvert > 0 \Rightarrow \lvert g(x_2) - l_g \rvert < mv_{\epsilon_2}))$	($\Rightarrow_E$ L_{39} L_{38})
L_{42}.	L_{42}	$\vdash 0 < c_{\delta_2} \wedge \forall x_2 \bullet (\lvert x_2 - a \rvert < c_{\delta_2} \wedge \lvert x_2 - a \rvert > 0 \Rightarrow \lvert g(x_2) - l_g \rvert < mv_{\epsilon_2})$	(Hyp)
L_{44}.	L_{42}	$\vdash 0 < c_{\delta_2}$	($\wedge$E-F L_{42})
L_{45}.	L_{42}	$\vdash \forall x_2 \bullet (\lvert x_2 - a \rvert < c_{\delta_2} \wedge \lvert x_2 - a \rvert > 0 \Rightarrow \lvert g(x_2) - l_g \rvert < mv_{\epsilon_2})$	($\wedge$E-F L_{42})
L_{46}.	L_{42}	$\vdash \lvert mv_{x_2} - a \rvert < c_{\delta_2} \wedge \lvert mv_{x_2} - a \rvert > 0 \Rightarrow \lvert g(mv_{x_2}) - l_g \rvert < mv_{\epsilon_2}))$	($\forall$E-F L_{45})
L_{11}.	L_{11}	$\vdash \lvert c_x - a \rvert > 0 \wedge \lvert c_x - a \rvert < mv_\delta$	(Hyp)
L_{14}.	L_{11}	$\vdash \lvert c_x - a \rvert > 0$	($\wedge$E-F L_{11})
L_{13}.	L_{11}	$\vdash \lvert c_x - a \rvert < mv_\delta$	($\wedge$E-F L_{11})
L_5.	L_5	$\vdash 0 < c_\epsilon$	(Hyp)
L_{61}.	$\mathcal{H}_1$	$\vdash 0 \leq 0$	(AskCS-B)
L_{59}.	$\mathcal{H}_1$	$\vdash mv_\delta \leq c_{\delta_1}$	(TellCS-B)
L_{57}.	$\mathcal{H}_2$	$\vdash 0 \leq 0$	(AskCS-B)
L_{55}.	$\mathcal{H}_2$	$\vdash mv_\delta \leq c_{\delta_2}$	(TellCS-B)
L_{52}.	$\mathcal{H}_2$	$\vdash mv_{x_2} = c_x$	(TellCS-B)

$L_{53}.$	$\mathcal{H}_2$	$\vdash mv_{\epsilon_2} \leq \frac{1}{2} * c_\epsilon$	(TELLCS-B)
$L_{50}.$	$\mathcal{H}_2$	$\vdash \lvert mv_{x_2} - a\rvert < c_{\delta_2}$	(SOLVE*-B L_{13} L_{55})
$L_{51}.$	$\mathcal{H}_2$	$\vdash \lvert mv_{x_2} - a\rvert > 0$	(SOLVE*-B L_{14} L_{57})
$L_{47}.$	$\mathcal{H}_2$	$\vdash \lvert mv_{x_2} - a\rvert < c_{\delta_2} \wedge \lvert mv_{x_2} - a\rvert > 0$	($\wedge$I-B L_{50} L_{51})
$L_{49}.$	$\mathcal{H}_2$	$\vdash \lvert g(mv_{x_2}) - l_g\rvert < mv_{\epsilon_2}$	($\Rightarrow_E$ L_{47} L_{46})
$L_{48}.$	$\mathcal{H}_2$	$\vdash \lvert g(c_x) - l_g\rvert < \frac{1}{2} * c_\epsilon$	(SOLVE*-B L_{49} L_{52} L_{53})
$L_{43}.$	$\mathcal{H}_2$	$\vdash \lvert g(c_x) - l_g\rvert < \frac{1}{2} * c_\epsilon$	($\Rightarrow$E-F L_{47} L_{46} L_{48})
$L_{40}.$	$\mathcal{H}_1$	$\vdash \lvert g(c_x) - l_g\rvert < \frac{1}{2} * c_\epsilon$	($\exists$E-F L_{41} L_{43})
$L_{37}.$	$\mathcal{H}_1$	$\vdash \lvert g(c_x) - l_g\rvert < \frac{1}{2} * c_\epsilon$	($\Rightarrow$E-F L_{39} L_{38} L_{40})
$L_{31}.$	$\mathcal{H}_1$	$\vdash \lvert 1\rvert \leq mv$	(TELLCS-B)
$L_{32}.$	$\mathcal{H}_1$	$\vdash mv_{\epsilon_1} \leq \frac{c_\epsilon}{2*mv}$	(TELLCS-B)
$L_{33}.$	$\mathcal{H}_1$	$\vdash \lvert g(c_x) - l_g\rvert < \frac{c_\epsilon}{2}$	(SIMPLIFY-B L_{37})
$L_{34}.$	$\mathcal{H}_1$	$\vdash 0 < mv$	(TELLCS-B)
$L_{35}.$	$\mathcal{H}_1$	$\vdash mv_{x_1} = c_x$	(TELLCS-B)
$L_{29}.$	$\mathcal{H}_1$	$\vdash \lvert mv_{x_1} - a\rvert < c_{\delta_1}$	(SOLVE*-B L_{13} L_{59})
$L_{30}.$	$\mathcal{H}_1$	$\vdash \lvert mv_{x_1} - a\rvert > 0$	(SOLVE*-B L_{14} L_{61})
$L_{26}.$	$\mathcal{H}_1$	$\vdash \lvert mv_{x_1} - a\rvert < c_{\delta_1} \wedge \lvert mv_{x_1} - a\rvert > 0$	($\wedge$I-B L_{29} L_{30})
$L_{28}.$	$\mathcal{H}_1$	$\vdash \lvert f(mv_{x_1}) - l_f\rvert < mv_{\epsilon_1}$	($\Rightarrow_E$ L_{26} L_{25})
$L_{27}.$	$\mathcal{H}_1$	$\vdash \lvert ((f(c_x) + g(c_x)) - l_f) - l_g\rvert < c_\epsilon$	(COMPLEXESTIMATE-B L_{28} L_{31} L_{32} L_{33} L_{34} L_{35})
$L_{22}.$	$\mathcal{H}_1$	$\vdash \lvert ((f(c_x) + g(c_x)) - l_f) - l_g\rvert < c_\epsilon$	($\Rightarrow$E-F L_{26} L_{25} L_{27})
$L_{19}.$	$\mathcal{H}_3$	$\vdash \lvert ((f(c_x) + g(c_x)) - l_f) - l_g\rvert < c_\epsilon$	($\exists$E-F L_{20} L_{21})
$L_{16}.$	$\mathcal{H}_3$	$\vdash \lvert ((f(c_x) + g(c_x)) - l_f) - l_g\rvert < c_\epsilon$	($\Rightarrow$E-F L_{18} L_{17} L_{19})
$L_{12}.$	$\mathcal{H}_3$	$\vdash \lvert (f(c_x) + g(c_x)) - (l_f + l_g)\rvert < c_\epsilon$	(SIMPLIFY-B L_{16})
$L_{10}.$	$\mathcal{H}_4$	$\vdash \lvert c_x - a\rvert < mv_\delta \wedge \lvert c_x - a\rvert > 0$ $\Rightarrow \lvert (f(c_x) + g(c_x)) - (l_f + l_g)\rvert < c_\epsilon$	($\Rightarrow$I-B L_{12})
$L_9.$	$\mathcal{H}_4$	$\vdash \forall x.(\lvert x - a\rvert < mv_\delta \wedge \lvert x - a\rvert > 0$ $\Rightarrow \lvert (f(x) + g(x)) - (l_f + l_g)\rvert < c_\epsilon)$	($\forall$I-B L_{10})
$L_8.$	$\mathcal{H}_4$	$\vdash 0 < mv_\delta$	(TELLCS-B)
$L_7.$	$\mathcal{H}_4$	$\vdash 0 < mv_\delta \wedge \forall x.(\lvert x-a\rvert < mv_\delta \wedge \lvert x-a\rvert > 0$ $\Rightarrow \lvert (f(x) + g(x)) - (l_f + l_g)\rvert < c_\epsilon)$	($\wedge$I-B L_8 L_9)
$L_6.$	$\mathcal{H}_4$	$\vdash \exists \delta.(0 < \delta \wedge \forall x.(\lvert x-a\rvert < \delta \wedge \lvert x-a\rvert > 0$ $\Rightarrow \lvert (f(x) + g(x)) - (l_f + l_g)\rvert < c_\epsilon))$	($\exists$I-B L_7)
$L_4.$	Lim_f, Lim_g	$\vdash 0 < c_\epsilon \Rightarrow \exists \delta.(0 < \delta \wedge$ $\forall x.(\lvert x - a\rvert < \delta \wedge \lvert x - a\rvert > 0$ $\Rightarrow \lvert (f(x) + g(x)) - (l_f + l_g)\rvert < c_\epsilon))$	($\Rightarrow$I-B L_6)
$L_1.$	Lim_f, Lim_g	$\vdash \forall \epsilon.(0 < \epsilon \Rightarrow \exists \delta.(0 < \delta \wedge$ $\forall x.(\lvert x - a\rvert < \delta \wedge \lvert x - a\rvert > 0$ $\Rightarrow \lvert (f(x) + g(x)) - (l_f + l_g)\rvert < \epsilon)))$	($\forall$I-B L_4)
$LIM+.$	Lim_f, Lim_g	$\vdash \lim_{x \to a}(f(x) + g(x)) = l_f + l_g$	(DEFNUNFOLD-B L_1)

$\mathcal{H}_1 = \{Lim_f, Lim_g, L_5, L_{11}, L_{21}\}$, $\mathcal{H}_2 = \{Lim_f, Lim_g, L_5, L_{11}, L_{21}, L_{42}\}$

$\mathcal{H}_3 = \{Lim_f, Lim_g, L_5, L_{11}\}$, $\mathcal{H}_4 = \{Lim_f, Lim_g, L_5\}$

References

[1] A.J. Aho, J. Hopcroft, and J. Ullman, editors. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[2] P.B. Andrews. Transforming Matings into Natural Deduction Proofs. In W. Bibel and R.A. Kowalski, editors, *Proceedings of the 5th Conference on Automated Deduction (CADE–5)*, volume 87 of *LNCS*, pages 281–292, Les ArcsFrance, June7–9 1980. Springer VerlagGermany.

[3] P.B. Andrews, M. Bishop, S. Issar, D. Nesmith, F. Pfenning, and H. Xi. TPS: A Theorem Proving System for Classical Type Theory. *Journal of Automated Reasoning*, 16(3):321–353, 1996.

[4] S. Autexier. *Hierarchical Contextual Rewriting*. PhD thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 2003. To appear.

[5] P. Baumgartner and U. Furbach. PROTEIN, A PROver with a Theory INterface. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction (CADE–12)*, volume 814 of *LNAI*, pages 769–773, NancyFrance, June26–July1 1994. Springer VerlagGermany.

[6] C. Benzmüller, M. Bishop, and V. Sorge. Integrating TPS and OMEGA. *Journal of Universal Computer Science*, 5:188–207, 1999.

[7] W.W. Bledsoe. Challenge Problems in Elementary Analysis. *Journal of Automated Reasoning*, 6:341–359, 1990.

[8] W.W. Bledsoe, R.S. Boyer, and W.H. Henneman. Computer Proofs of Limit Theorems. *Artificial Intelligence*, 3(1):27–60, 1972.

[9] A. Bundy. The Use of Explicit Plans to Guide Inductive Proofs. In E.L. Lusk and R.A. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction (CADE–9)*, volume 310 of *LNCS*, pages 111–120, Argonne, Illinois, USA, 1988. Springer VerlagGermany.

[10] J.G. Carbonell. Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition. In R.S. Michalsky, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 371–392. Morgan Kaufmann Publ., Los Altos, 1986.

[11] L. Cheikhrouhou and J. Siekmann. Planning Diagonalization Proofs. In F. Giunchiglia, editor, *Artificial Intelligence: Methodology, Systems and Applications, Proceedings of the of the 8th International Conference (AIMSA'98)*, volume 1480 of *LNAI*, pages 167–180, Sozopol, Bulgaria, September21–23 1998. Springer VerlagGermany.

[12] L. Cheikhrouhou and V. Sorge. $\mathcal{PDS}$ — A Three-Dimensional Data Structure for Proof Plans. In *Proceedings of the International Conference on Artificial and Computational Intelligence for Decision, Control and Automation in Engineering and Industrial Applications (ACIDCA'2000)*, Monastir, Tunisia, March22–24 2000.

[13] A. Cohen, S. Murray, M. Pollet, and V. Sorge. Certifying solutions to permutation group problems. In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction (CADE–19)*, number 2741 in LNAI, Miami Beach, Fl, USA, 2003. Springer VerlagGermany.

[14] R.L. Constable, S.F. Allen, H.M. Bromley, R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the* NUPRL *Proof Development System*. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[15] Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA.

[16] H. de Nivelle. *The Bliksem Theorem Prover, Version 1.12*. Max-Plank-Institut, Im Stadtwald, Saarbruecken, Germany, October1999. available `http://www.mpi-sb.mpg.de/~bliksem/manual.ps`.

[17] M. Drummond. On precondition achievement and the computational economics of automatic planning. In C. Bäckström and E. Sandwall, editors, *Current Trends in AI Planning*. IOS Press, 1994.

[18] R. Engelmore and T. Morgan, editors. *Blackboard Systems*. Addison-Wesley, 1988.

[19] L.D. Erman, P. London, and S. Fickas. The Design and an Example Use of HEARSAY-III. In B. Buchanan, editor, *Proceedings of the 6th International Joint Conference on Artificial Intelligence (ICJAI)*, pages 409–415, Tokyo, Japan, August20–23 1979. Morgan Kaufmann.

[20] R.E. Fikes, P.E. Hart, and N.J. Nilsson. Some New Directions in Robot Problem Solving. *Machine Intelligence*, 7, 1971.

[21] R.E. Fikes and N.J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:189–208, 1971.

[22] H. Ganzinger, editor. *Proceedings of the 16th International Conference on Automated Deduction (CADE–16)*, volume 1632 of *LNAI*, Trento, Italy, July7–10, 1999. Springer Verlag-Germany.

[23] G. Gentzen. Untersuchungen über das Logische Schließen I und II. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

[24] M.L. Ginsberg. Dynamic Backtracking. *Journal of Artificial Intelligence Research*, 1:25—46, 1993.

[25] M.J.C. Gordon and T.F. Melham. *Introduction to HOL*. Cambridge University Press, Cambridge, UK, 1993.

[26] B. Hayes-Roth. A Blackboard Architecture for Control. *Artificial Intelligence*, 25:251–321, 1985.

[27] T. Hillenbrand, A. Jaeger, and B. Löchner. System Description: WALDMEISTER, Improvements in Performance and Ease of Use. In Ganzinger [22], pages 232 – 236.

[28] S. Kambhampati. Formalizing Dependency Directed Backtracking and Explanation-based Learning in Refinement Search. In W.J. Clancey and D. Weld, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96) and Eighth Innovative Applications of Artificial Intelligence Conference (IAAI-96)*, pages 757–762, Portland, Oregon, USA, August4–8 1996. AAAI Press, Menlo Park, CA, USA.

[29] H. Kirchner and C. Ringeissen, editors. *Proceedings of Third International Workshop on Frontiers of Combining Systems (FROCOS 2000)*, volume 1794 of *LNCS*, NancyFrance, March22–24 2000. Springer VerlagGermany.

[30] U. Kühler. *A Tactic-Based Inductive Theorem Prover for Data Types with Partial Operations*. PhD thesis, Sankt Augustin, 2000.

[31] W. McCune. Otter 3.0 Reference Manual and Guide. Technical Report ANL-94-6, Argonne National Laboratory, Argonne, Illinois 60439, USA, 1994.

[32] W. McCune. Solution of the Robbins Problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[33] A. Meier. TRAMP: Transformation of Machine-Found Proofs into Natural Deduction Proofs at the Assertion Level. In D. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE–17)*, volume 1831 of *LNAI*, pages 460–464, Pittsburgh, PA, USA, June17–20 2000. Springer VerlagGermany.

[34] A. Meier. MULTI – *Proof Planning with Multiple Strategies*. PhD thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 2004.

[35] A. Meier and E. Melis. Proof planning limit problems with multiple strategies. Seki Report SR-2004-04, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany, 2004.

[36] A. Meier, M. Pollet, and V. Sorge. Classifying Isomorphic Residue Classes. In R. Moreno-Díaz, B. Buchberger, and J.L. Freire, editors, *Proceedings of the 8th International Workshop on Computer Aided Systems Theory (EuroCAST 2001)*, volume 2178 of *LNCS*, pages 494–508, Las Palmas de Gran Canaria, Spain, February19–23 2001. Springer Verlag, Germany.

[37] A. Meier, M. Pollet, and V. Sorge. Classifying Residue Classes – Results of a Case Study. Seki Report SR-01-01, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany, 2001.

[38] A. Meier, M. Pollet, and V. Sorge. Comparing Approaches to Explore the Domain of Residue Classes. *Journal of Symbolic Computation, Special Issue on the Integration of Automated Reasoning and Computer Algebra Systems*, 34(4):287–306, 2002. S. Linton and R. Sebastiani, eds.

[39] A. Meier and V. Sorge. Exploring Properties of Residue Classes. In M. Kerber and M. Kohlhase, editors, *Symbolic Computation and Automated Reasoning – The CALCULEMUS-2000 Symposium*, pages 175–190, St. Andrews, UK, August6–7, 2000 2001. AK Peters, Natick, MA, USA.

[40] Andreas Meier, Erica Melis, and Martin Pollet. Adaptable mixed-initiative proof planning for educational interaction. *Electronic Notes in Theoretical Computer Science*, 2004. To appear.

[41] E. Melis. AI-Techniques in Proof Planning. In H. Prade, editor, *Proceedings of of the 13th European Conference on Artifical Intelligence*, pages 494–498, Brighton, UK, August23–28 1998. John Wiley & Sons, Chichester, UK.

[42] E. Melis. The "Limit" Domain. In R. Simmons, M. Veloso, and S. Smith, editors, *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, pages 199–206, Pittsburgh, PEN, USA, June7–10 1998. AAAI Press, Menlo Park, CA, USA.

[43] E. Melis, J. Buedenbender, E. Andres, A. Frischauf, G. Goguadse, P. Libbrecht, M. Pollet, and C. Ullrich. ACTIVEMATH: A generic and adaptive web-based learning environment. *Artificial Intelligence and Education*, 12(4):385–407, 2001.

[44] E. Melis and A. Meier. Proof Planning with Multiple Strategies. In J. Loyd, V. Dahl, U. Furbach, M. Kerber, K. Lau, C. Palamidessi, L.M. Pereira, and Y. Sagivand P. Stuckey, editors, *First International Conference on Computational Logic (CL-2000)*, volume 1861 of *LNAI*, pages 644–659, London, UK, 2000. Springer-Verlag.

[45] E. Melis and J. Siekmann. Knowledge-Based Proof Planning. *Artificial Intelligence*, 115(1):65–105, 1999.

[46] E. Melis and C. Ullrich. Flexibly Interleaving Processes. In K.-D. Althoff and R. Bergmann, editors, *International Conference on Case-Based Reasoning*, volume 1650 of *LNAI*, pages 263–275. Springer VerlagGermany, 1999.

[47] E. Melis, J. Zimmer, and T. Müller. Integrating Constraint Solving into Proof Planning. In Kirchner and Ringeissen [29], pages 32–46.

[48] R. Milner. The use of machines to assist in rigorous proof. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 77–88. Prentice-Hall, 1984.

[49] S. Minton. Explanation-Based Learning: A Problem Solving Perspective. *Artificial Intelligence*, 40:63–118, 1989.

[50] A. Newell and H.A. Simon. GPS: a Program that Simulates Human Thought. In E.A. Feigenbaum and J. Feldmann, editors, *Computers and Thought*. McGraw-Hill, 1963.

[51] L. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in LNCS. Springer Verlag-Germany, 1994.

[52] D. Redfern. *The Maple Handbook: Maple V Release 5*. Springer VerlagGermany, 1999.

[53] E. Rich and K. Knight, editors. *Artificial Intelligence*. McGraw-Hill, 1991.

[54] S. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach*. Prentice Hall, Englewood Cliffs, 1995.

[55] S. Scholl. Hierarchische Analogie im Beweisplanen. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbücken, 2003.

[56] J. Siekmann, C. Benzmüller, V. Brezhnev, L. Cheikhrouhou, A. Fiedler, A. Franke, H. Horacek, M. Kohlhase, A. Meier, E. Melis, M. Moschner, I. Normann, M. Pollet, V. Sorge, C. Ullrich, C.P. Wirth, and J. Zimmer. Proof Development with OMEGA. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE–18)*, number 2392 in LNAI, pages 144–149, Kopenhagen, Denmark, 2002. Springer Verlag-Germany.

[57] J. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, I. Normann, and M. Pollet. Proof Development in OMEGA: The Irrationality of Square Root of 2. In Fairouz Kamareddine, editor, *Thirty Five Years of Automating Mathematics*, Kluwer Applied Logic series. Kluwer Academic Publishers, 2003. In Print.

[58] V. Sorge. Non-Trivial Symbolic Computations in Proof Planning. In Kirchner and Ringeissen [29], pages 121–135.

[59] R.M. Stallmann and G.J. Sussmann. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2), 1977.

[60] A. Tate. Generating Project Networks. In R. Reddy, editor, *Proceedings of the 5th International Joint Conference on Artificial Intelligence (ICJAI)*, pages 888–893, Cambridge, MA, USA, August22–25 1977. Morgan Kaufmann, San Mateo, CA, USA.

[61] C. Ullrich. Analogie im Beweisplanen. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbücken, 2000.

[62] M.M. Veloso, J. Carbonell, M.A. Perez, D. Borrajo, E. Fink, and J. Blythe. Integrating Planning and Learning: The Prodigy Architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120, 1995.

[63] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, Th. Engel, E. Keen, C. Theobalt, and D. Topic. System Description: Spass Version 1.0.0. In Ganzinger [22], pages 378–382.

[64] D.S. Weld. An Introduction to Least Commitment Planning. *AI Magazine*, 15(4):27–61, 1994.

[65] J. Zimmer. Constraintlösen für Beweisplanung. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 2000.